

Quickest Visibility Queries in Polygonal Domains

Haitao Wang

Department of Computer Science
Utah State University, Logan, UT 84322, USA
haitao.wang@usu.edu

Abstract. Let s be a point in a polygonal domain \mathcal{P} of $h - 1$ holes and n vertices. We consider a *quickest visibility query* problem. Given a query point q in \mathcal{P} , the goal is to find a shortest path in \mathcal{P} to move from s to see q as quickly as possible. Previously, Arkin et al. (SoCG 2015) built a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(K \log^2 n)$ time, where $\alpha(n)$ is the inverse Ackermann function and K is the size of the visibility polygon of q in \mathcal{P} (and K can be $\Theta(n)$ in the worst case). In this paper, we present a new data structure of size $O(n \log h + h^2)$ that can answer each query in $O(h \log h \log n)$ time. Our result improves the previous work when h is relatively small. In particular, if h is a constant, then our result even matches the best result for the simple polygon case (i.e., $h = 1$), which is optimal. As a by-product, we also have a new algorithm for a *shortest-path-to-segment query* problem. Given a query line segment τ in \mathcal{P} , the query seeks a shortest path from s to all points of τ . Previously, Arkin et al. gave a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(\log^2 n)$ time, and another data structure of size $O(n^3 \log n)$ with $O(\log n)$ query time. We present a data structure of size $O(n)$ with query time $O(h \log \frac{n}{h})$, which also favors small values of h and is optimal when $h = O(1)$.

1 Introduction

Let \mathcal{P} be a polygonal domain with $h - 1$ holes and a total of n vertices, i.e., there is an outer simple polygon containing $h - 1$ pairwise disjoint holes and each hole itself is a simple polygon. If $h = 1$, then \mathcal{P} becomes a simple polygon. For any two points s and t in \mathcal{P} , a *shortest path* from s to t is a path in \mathcal{P} connecting s and t with the minimum Euclidean length. Two points p and q are *visible* to each other if the line segment \overline{pq} is in \mathcal{P} . For any point q in \mathcal{P} , its *visibility polygon* consists of all points of \mathcal{P} visible to q , denoted by $Vis(q)$.

We consider the following *quickest visibility query* problem. Let s be a source point in \mathcal{P} . Given any point q in \mathcal{P} , the query asks for a path to move from s to see q as quickly as possible. Such a “quickest path” is actually a shortest path from s to any point of $Vis(q)$. The problem has been recently studied by Arkin et al. [1], who built a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(K \log^2 n)$ time, where K is the size of $Vis(q)$. In this paper, we present a new data structure of $O(n \log h + h^2)$ size with $O(h \log h \log n)$ query time. Our result improves the previous work when h is relatively small. Interesting is that the query time is independent of K , which can be $\Theta(n)$ in the worst case. Our result is also interesting in that when $h = O(1)$, the data structure has $O(n)$ size and $O(\log n)$ query time, which even matches the best result for the simple polygon case [1] and is optimal.

As in [1], in order to solve the quickest visibility queries, we also solve a *shortest-path-to-segment query* problem (or *segment query* for short), which may have independent interest. Given any line segment τ in \mathcal{P} , the segment query asks for a shortest path from s to all points of τ . Arkin et al. [1] gave a data structure of size $O(n^2 2^{\alpha(n)} \log n)$ that can answer each query in $O(\log^2 n)$ time, and another data structure of size $O(n^3 \log n)$ with $O(\log n)$ query time. We present a new data structure of $O(n)$ size with $O(h \log \frac{n}{h})$ query time. Our result again favors small values of h and

attains optimality when $h = O(1)$, which also matches the best result for the simple polygon case [1,13].

Given the shortest path map of s , our quickest visibility query data structure can be built in $O(n \log h + h^2 \log h)$ time and our segment query data structure can be built in $O(n)$ time. Arkin et al.’s quickest visibility query data structure and their first segment query data structure can both be built in $O(n^2 2^{\alpha(n)} \log n)$ time, and their second segment query data structure can be built in $O(n^3 \log n)$ time [1].

Throughout the paper, whenever we talk about a query related to paths in \mathcal{P} , the query time always refers to the time for computing the path length, and to output the actual path, it needs additional time linear in the number of edges of the path by standard techniques (we will omit the details about this).

1.1 Related Work

The traditional shortest path query problem has been studied extensively, which is to compute a shortest path to move from s to “reach” a query point. Each shortest path query can be answered in $O(\log n)$ time by using the *shortest path map* of s , denoted by $SPM(s)$, which is of $O(n)$ size. To build $SPM(s)$, Mitchell [28] gave an algorithm of $O(n^{3/2+\epsilon})$ time for any $\epsilon > 0$ and $O(n)$ space, and later Hershberger and Suri [22] presented an algorithm of $O(n \log n)$ time and space. If \mathcal{P} is a simple polygon (i.e., $h = 1$), $SPM(s)$ can be built in $O(n)$ time, e.g., see [17].

For the quickest visibility queries, Arkin et al. [1] also built a “quickest visibility map” of $O(n^7)$ size in $O(n^8 \log n)$ time, which can answer each query in $O(\log n)$ time. In addition, Arkin et al. [1] gave a conditional lower bound on the problem by showing that the 3SUM problem on n numbers can be solved in $O(\tau_1 + n \cdot \tau_2)$ time, where τ_1 is the preprocessing time and τ_2 is the query time. Therefore, a data structure of $o(n^2)$ preprocessing time and $o(n)$ query time would lead to an $o(n^2)$ time algorithm for 3SUM.

In the simple polygon case (i.e., $h = 1$), better results are possible for both the quickest visibility queries and the segment queries. For the quickest visibility queries, Khosravi and Ghodsi [24] first proposed a data structure of $O(n^2)$ size that can answer each query in $O(\log n)$ time. Arkin et al. [1] gave an improved result and they built a data structure of $O(n)$ size in $O(n)$ time, with $O(\log n)$ query time. For the segment queries, Arkin et al. [1] built a data structure of $O(n)$ size in $O(n)$ time, with $O(\log n)$ query time. Chiang and Tamassia [13] achieved the same result for the segment queries and they also gave some more general results (e.g., when the query is a convex polygon).

Similar in spirit to the “point-to-segment” shortest path problem, Cheung and Daescu [12] considered a “point-to-face” shortest path problem in 3D and approximation algorithms were given for the problem.

1.2 Our Techniques

We first propose a decomposition \mathcal{D} of \mathcal{P} by $O(h)$ shortest paths from s to certain vertices of $SPM(s)$. The decomposition \mathcal{D} , whose size is $O(n)$, has $O(n)$ cells with the following three key properties. First, any segment τ in \mathcal{P} can intersect at most $O(h)$ cells of \mathcal{D} . Second, for each cell Δ of \mathcal{D} , $\tau \cap \Delta$ consists of at most two sub-segments of τ . Third, after $O(n)$ time preprocessing, for each sub-segment τ' of τ in any cell of \mathcal{D} , the shortest path from s to τ' can be computed in $O(\log n)$ time. With \mathcal{D} , we can easily answer each segment query in $O(h \log \frac{n}{h})$ time by a “pedestrian” algorithm.

To solve the quickest visibility queries, an observation is that the shortest path from s to see q is a shortest path from s to a *window* of $\text{Vis}(q)$, i.e., an extension of the segment \overline{qu} for some reflex vertex u of \mathcal{P} . Hence, the query can be answered by calling segment queries on all $O(K)$ windows of $\text{Vis}(s)$ and returning the shortest path. This leads to the $O(K \log^2 n)$ time query algorithm in [1].

If we follow the same algorithmic scheme and using our new segment query algorithm, then we would obtain an algorithm of $O(K \cdot h \cdot \log \frac{n}{h})$ time for the quickest visibility queries. We instead present a “smarter” algorithm. We propose a “pruning algorithm” that prunes some “unnecessary” portions of the windows such that it suffices to consider the remaining parts of the windows. Further, with the help of the decomposition \mathcal{D} , we show that a shortest path from s to the remaining windows can be found in $O((K + h) \log h \log n)$ time. We refer to it as *the preliminary result*. To achieve this result, we solve many other problems, which may be of independent interest. For example, we build a data structure of $O(n \log h)$ size such that given any query point t and line segment τ in \mathcal{P} , we can compute in $O(\log h \log n)$ time the intersection between τ and the shortest path from s to t in \mathcal{P} (or report none if they do not intersect). Our above pruning algorithm is based on a new and interesting technique of using “bundles”.

To further reduce the query time to $O(h \log h \log n)$, the key idea is that by using the extended corridor structure of \mathcal{P} [8,11], we show that there exists a set $\mathcal{S}(q)$ of $O(h)$ *candidate windows* such that a shortest path from s to see the query point q must be a shortest path from s to a window in $\mathcal{S}(q)$. This is actually quite consistent with the result in the simple polygon case, where only one window is needed for answering each quickest visibility query [1]. Once the set $\mathcal{S}(q)$ is computed, we can apply our pruning algorithm discussed above on $\mathcal{S}(q)$ to answer the quickest visibility query in additional $O(h \log h \log n)$ time. To compute $\mathcal{S}(q)$, we give an algorithm of $O(h \log n)$ time, without having to explicitly compute $\text{Vis}(s)$. The algorithm is based on a modification of the algorithm given in [9] that can compute $\text{Vis}(q)$ in $O(K \log n)$ time for any point q , after $O(n + h^2)$ space and $O(n + h^2 \log h)$ time preprocessing.

The rest of the paper is organized as follows. In Section 2, we introduce notation and review some concepts. In Section 3, we introduce the decomposition \mathcal{D} of \mathcal{P} , and present our algorithm for the segment queries. We present our preliminary result for the quickest visibility queries in Section 4 and give the improved result in Section 5. Section 6 concludes the paper.

2 Preliminaries

For any subset A of \mathcal{P} , we say that a point p is (*weakly*) *visible* to A if p is visible to at least one point of A . For any point $t \in \mathcal{P}$, we use $\pi(s, t)$ to denote a shortest path from s to t in \mathcal{P} , and in the case where the shortest path is not unique, $\pi(s, t)$ may refer to an arbitrary such path. With a little abuse of notation, for any subset A of \mathcal{P} , we use $\pi(s, A)$ to denote a shortest path from s to all points of A ; we use $d(s, A)$ to denote the length of $\pi(s, A)$, i.e., $d(s, A) = \min_{t \in A} d(s, t)$.

Let \mathcal{V} denote the set of all vertices of \mathcal{P} .

The shortest path map $\text{SPM}(s)$. $\text{SPM}(s)$ is a decomposition of \mathcal{P} into regions (or cells) such that in each cell σ , the sequence of obstacle vertices along $\pi(s, t)$ is fixed for all t in σ [22,28]. Further, the *root* of σ , denoted by $r(\sigma)$, is the last vertex of $\mathcal{V} \cup \{s\}$ in $\pi(s, t)$ for any point $t \in \sigma$ (hence $\pi(s, t) = \pi(s, r(\sigma)) \cup \overline{r(\sigma)t}$; note that $r(\sigma)$ is s if s is visible to t). We classify each edge of a cell σ into three types: a portion of an edge of \mathcal{P} , an *extension segment*, which is a line segment extended from $r(\sigma)$ along the opposite direction from $r(\sigma)$ to the vertex of $\pi(s, t)$ preceding $r(\sigma)$, and a

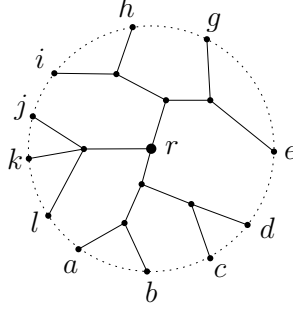


Fig. 1. Illustrating a planar tree T with root r : a is a base leaf and the list $\mathcal{L}_i(T, a)$ is a, b, c, \dots, l .

bisector curve/edge that is a hyperbolic arc. For each point t on a bisector edge of $SPM(s)$, t is on the common boundary of two cells and there are two different shortest paths from s to t through the roots of the two cells, respectively. The *vertices* of $SPM(s)$ include $\mathcal{V} \cup \{s\}$ and all intersections of edges of $SPM(s)$. The intersection of two bisector edges is called a *triple point*, which has more than two shortest paths from s . The map $SPM(s)$ has $O(n)$ vertices, edges, and cells [22,28].

For differentiation, we call the vertices and edges of the polygonal domain \mathcal{P} the *obstacle vertices* and the *obstacle edges*, respectively. The holes and the outer polygon of \mathcal{P} are also called *obstacles*.

The *shortest path tree* $SPT(s)$ is the union of shortest paths from s to all obstacle vertices of \mathcal{P} . $SPT(s)$ has $O(n)$ edges [22,28]. Given $SPM(s)$, $SPT(s)$ can be obtained in linear time. We sometimes consider a further decomposition of $SPM(s)$ by having all edges of $SPT(s)$ in it.

For ease of exposition, we make a general position assumption that no obstacle vertex has more than one shortest path from s and no point of \mathcal{P} has more than three shortest paths from s . Hence, no bisector edge of $SPM(s)$ intersects an obstacle vertex and no three bisector edges intersect at the same point.

For any polygon P , we use $|P|$ to denote the number of vertices of P and use ∂P to denote the boundary of P .

Ray-shooting queries in simple polygons. Let P be a simple polygon. With $O(|P|)$ time and space preprocessing, each ray-shooting query in P (i.e., given a ray in P , find the first point on ∂P hit by the ray) can be answered in $O(\log |P|)$ time [6,21]. The result can be extended to curved simple polygons or splinegons [26].

The canonical lists and cycles of planar trees. We will often talk about certain planar trees in \mathcal{P} (e.g., $SPT(s)$). Consider a tree T with root r . A leaf v is called a *base leaf* if it is the leftmost leaf of a subtree rooted at a child of r (e.g., see Fig. 1). Denote by $\mathcal{L}(T, v)$ the post-order traversal list of T starting from such a base leaf v , and we call it a *canonical list* of T . The root r must be the last node in $\mathcal{L}(T, v)$. We remove r from $\mathcal{L}(T, v)$ and make the remaining list a cycle by connecting its rear to its front, and let $\mathcal{C}(T)$ denote the circular list. Although T may have multiple base leaves, $\mathcal{C}(T)$ is unique and we call $\mathcal{C}(T)$ the *canonical cycle* of T . We further use $\mathcal{L}_i(T, v)$ (e.g., see Fig. 1) to denote the list of the leaves of T following their relative order in $\mathcal{L}(T, v)$ and use $\mathcal{C}_i(T)$ to denote the circular list of $\mathcal{L}_i(T, v)$. One reason we introduce these notation is the following. Let e be any edge of T . All nodes of T whose paths to r in T contain e must be consecutive in $\mathcal{L}(T, v)$ and $\mathcal{C}(T)$. Similarly, all leaves of T whose paths to r in T contain e must be consecutive in $\mathcal{L}_i(T, v)$ and $\mathcal{C}_i(T)$.

The following observation on shortest paths will be frequently referred to in the paper.

- Observation 1** 1. Suppose π_1 and π_2 are two shortest paths from s to two points in \mathcal{P} , respectively; then π_1 and π_2 do not cross each other.
2. Suppose π_1 is a shortest path from s to a point in \mathcal{P} and τ is a line segment in \mathcal{P} ; then the intersection of π_1 and τ is a sub-segment of τ (which may be a single point or empty).

3 The Decomposition \mathcal{D} and the Segment Queries

In this section, we introduce a decomposition \mathcal{D} of \mathcal{P} and use it to solve the segment query problem. The decomposition \mathcal{D} will also be useful for solving the quickest visibility queries.

We first define a set V of points. Let p be an intersection between a bisector edge of $SPM(s)$ and an obstacle edge. Since p is on a bisector edge, it is in two cells of $SPM(s)$ and has two shortest paths from s . We make two copies of p in the way that each copy belongs to only one cell (and thus corresponds to only one shortest path from s). We add the two copies of p to V . We do this for all intersections between bisector edges and obstacle edges. Consider a triple point p , which is in three cells of $SPM(s)$ and has three shortest paths from s . Similarly, we make three copies of p that belong to the three cells, respectively. We add the three copies of p to V . We do this for all triple points. This finishes the definition of V .

By definition, each point of V has exactly one shortest path from s . Let Π_V denote the set of shortest paths from s to all points of V . Let T_V be the union of all shortest paths of Π_V . We consider points of V distinct although some of them are copies of the same physical point. In this way, we can consider T_V as a “physical” tree rooted at s .

Definition 1. Define \mathcal{D} to be the decomposition of \mathcal{P} by the edges of T_V .

In the following, we assume the shortest path map $SPM(s)$ has already been computed. We have the following lemma about the decomposition \mathcal{D} .

- Lemma 1.** 1. The size of the set V is $O(h)$.
2. The combinatorial size of \mathcal{D} is $O(n)$.
3. Each cell of \mathcal{D} is simply connected.
4. For any segment τ in \mathcal{P} , τ can intersect at most $O(h)$ cells of \mathcal{D} . Further, for each cell Δ of \mathcal{D} , the intersection τ and Δ consists of at most two (maximal) sub-segments of τ .
5. After $O(n)$ time preprocessing, for any segment τ' in a cell Δ of \mathcal{D} , the shortest path from s to τ' can be computed in $O(\log |\Delta|)$ time, where $|\Delta|$ is the combinatorial size of Δ .
6. For each cell Δ of \mathcal{D} , Δ has at most two vertices r_1 and r_2 (both in $\mathcal{V} \cup \{s\}$), called “super-roots”, such that for any point $t \in \Delta$, $\pi(s, t)$ is the concatenation of $\pi(s, r)$ and the shortest path from r to t in Δ , for a super-root r in $\{r_1, r_2\}$.
7. Given the shortest path map $SPM(s)$, \mathcal{D} can be computed in $O(n)$ time.

We will prove Lemma 1 later in Section 3.2. Below we first give our data structure for answering segment queries by using Lemma 1.

3.1 The Segment Queries

As preprocessing, we first compute the decomposition \mathcal{D} . Then, we build a point location data structure on \mathcal{D} [14,25], which can be done in $O(n)$ time and $O(n)$ space since the size of \mathcal{D} is $O(n)$ by Lemma 1(2); the data structure can answer each point location query in $O(\log n)$ time.

In addition, for each cell Δ of \mathcal{D} , by Lemma 1(3), Δ is a simple polygon; we build a ray-shooting data structure on Δ [6,21]. Since the total size of all cells of \mathcal{D} is $O(n)$ by Lemma 1(2), the total preprocessing time and space for the ray-shooting queries on all cells of \mathcal{D} is $O(n)$.

Finally, we do the preprocessing in Lemma 1(5). Hence, given $SPM(s)$, the total preprocessing time and space is $O(n)$. The following lemma gives our query algorithm.

Lemma 2. *Given any segment τ in \mathcal{P} , we can compute a shortest path from s to τ in $O(h \log \frac{n}{h})$ time.*

Proof. Let a and b be the two endpoints of τ , respectively. Our algorithm works in a “pedestrian” way, as follows.

By using a point location query, we find the cell Δ_a of \mathcal{D} that contains a . Then, we check whether τ is contained in Δ_a . This can be done by using a ray-shooting query as follows. We shoot a ray ρ from a towards b and compute the first point p of $\partial\Delta_a$ hit by the ray. The segment τ is in Δ_a if and only if b is before p on the ray.

If τ is in Δ_a , then we can immediately compute the shortest path $\pi(s, \tau)$ from s to τ in $O(\log |\Delta_a|)$ time by Lemma 1(5).

Otherwise, we compute the shortest path $\pi(s, \overline{ap})$ from s to the sub-segment \overline{ap} of τ in $O(\log |\Delta_a|)$ time by Lemma 1(5). Next, based on the edge of \mathcal{D} containing p , we can determine in constant time the next cell Δ of \mathcal{P} that the ray ρ enters. We process the cell Δ in the similar way as the above for Δ_a . The algorithm finishes once we process a cell that contains b .

The above computes $\pi(s, \tau')$ for multiple sub-segments τ' of τ such that these sub-segments constitute exactly τ and each sub-segment is in a single cell of \mathcal{D} . Clearly, among all shortest paths from s to these sub-segments, the one with the minimum length is the shortest path from s to τ .

To analyze the running time of the above algorithm, let k be the number of the above sub-segments τ' of τ . Suppose $\tau'_1, \tau'_2, \dots, \tau'_k$ are these sub-segments ordered from a to b . For each $1 \leq i \leq k$, let Δ_i be the cell of \mathcal{D} that contains τ'_i . First of all, the point location query for a takes $O(\log n)$ time. For each $1 \leq i \leq k$, determining each sub-segment τ'_i needs a ray-shooting query in Δ_i , which takes $O(\log |\Delta_i|)$ time; computing the length of $\pi(s, \tau'_i)$ also takes $O(\log |\Delta_i|)$ time by Lemma 1(5). Hence, the total time of the algorithm is $O(\log n + \sum_{i=1}^k \log |\Delta_i|)$.

By Lemma 1(4), $k = O(h)$. Also, by Lemma 1(4), each cell may contain two of the above k sub-segments of τ , and thus it is possible that Δ_i and Δ_j refer to the same cell for $i \neq j$. Let S be the set of the distinct cells of Δ_i for $i = 1, 2, \dots, k$. Since each cell contains at most two of the above k sub-segments of τ , $\sum_{i=1}^k \log |\Delta_i| \leq 2 \cdot \sum_{\Delta \in S} \log |\Delta|$. Further, since the cells of S are distinct, we have $\sum_{\Delta \in S} |\Delta| = O(n)$. Due to $|S| \leq k = O(h)$, we have $\sum_{\Delta \in S} \log |\Delta| = O(h \log \frac{n}{h})$.

Therefore, the total time of the algorithm is bounded by $O(h \log \frac{n}{h})$. \square

We summarize our result for segment queries in the following theorem.

Theorem 1. *Given the shortest path map $SPM(s)$, we can build a data structure of $O(n)$ size in $O(n)$ time, such that each segment query can be answered in $O(h \log \frac{n}{h})$ time.*

3.2 The Decomposition \mathcal{D} and Proving Lemma 1

In this section we provide the details for \mathcal{D} and prove Lemma 1.

Let \mathcal{O} denote the obstacle space, which is the complement of the free space of \mathcal{P} . More specifically, \mathcal{O} consists of the $h - 1$ simple polygonal holes of \mathcal{P} and the (unbounded) region outside the

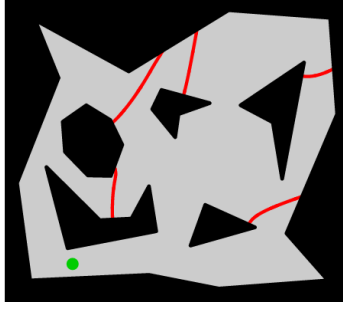


Fig. 2. Illustrating the bisector edges of shortest path map (the back area is the obstacle space): the green point is the source s and the red curves are the bisector edges. The figure is generated by the applet in [20]

outer boundary of \mathcal{P} . Let \mathcal{B} denote the union of all bisector edges of $SPM(s)$. Mitchell [27] proved that $\mathcal{O} \cup \mathcal{B}$ is simply connected and $\mathcal{P} \setminus \mathcal{B}$ is also simply connected (e.g., see Fig. 2). We consider $\mathcal{O} \cup \mathcal{B}$ as a planar graph G , defined as follows.

The vertex set of G consists of all obstacles of \mathcal{O} and all triple points of $SPM(s)$. For any two vertices of G , if they are connected by a chain of bisector edges in $SPM(s)$ such that the chain does not contain any other vertex of G , then G has an edge connecting the two vertices, and further, we call the above chain of bisector edges a *bisector super-curve* (e.g., in Fig. 2, each red curve is a bisector super-curve). We have the following observation about G .

Observation 2 G is a simple graph, i.e., G does not have a self-loop and no two vertices have more than one edge. G has $O(h)$ vertices, edges, and faces.

Proof. The first part of the observation can be proved easily from Mitchell’s observation in [27] that $\mathcal{P} \setminus \mathcal{B}$ is simply connected, as follows.

Indeed, assume to the contrary that G has a self-loop at a vertex v . According to our definition, the self-loop corresponds to a bisector super-curve that connects the vertex v (either a triple point or an obstacle) to itself. Let R be region bounded by bisector-super curve and v . Hence, R is closed, which contradicts with that $\mathcal{P} \setminus \mathcal{B}$ is simply connected.

Similarly, assume to the contrary that two vertices u and v have two edges. Then, the two edges correspond to two bisector super-curves. Thus, the region bounded by the two bisector super-curves and the two vertices is closed, incurring contradiction again.

To prove the second part of the observation, note that G is a planar graph.

First, it is known that the number of triple points is $O(h)$ [15]. Since there are h obstacles in \mathcal{O} , the number of vertices of G is $O(h)$.

Second, the faces of G correspond exactly to the faces of the $(\leq 1) - SPM$ of \mathcal{P} defined in [15], whose total number is proved to be $O(h)$ [15] (see Lemma 4.3 with $k = 1$). Therefore, the number of faces of G is $O(h)$.

Finally, since both the number of vertices and the number of faces of G are $O(h)$, the number of edges of G is also $O(h)$. \square

Let V_1 be the set of all triple points. It is known that $|V_1| = O(h)$ [15]. Let V_2 be the set of intersections between obstacle edges and bisector edges of $SPM(s)$. It is not difficult to see that each point of V_2 corresponds to an intersection between an obstacle and a bisector super-curve. Since G has $O(h)$ edges, there are $O(h)$ bisector super-curves. Thus, $|V_2| = O(h)$. Recall that V

consists of three copies of each point of V_1 and two copies of each vertex of V_2 . Since both $|V_1|$ and $|V_2|$ are $O(h)$, we have $|V| = O(h)$. This proves Lemma 1(1).

Since $|V| = O(h)$, Π_V is the set of $O(h)$ shortest paths. Note that each edge of any path of Π_V except the last edge (i.e., the one connecting a point of V) is an edge of the shortest path tree $SPT(s)$. Hence, the total number of edges of the tree T_V is $O(n)$. Since \mathcal{D} is the decomposition of \mathcal{P} by the edges of T_V , the combinatorial size of \mathcal{D} is $O(n)$. This proves Lemma 1(2).

Throughout the paper, let $h^* = |V|$. Hence, $h^* = O(h)$.

To prove the rest of Lemma 1, we introduce another decomposition \mathcal{D}' as follows.

Definition 2. Define \mathcal{D}' to be the decomposition of \mathcal{P} by the edges of $T_V \cup \mathcal{B}$.

By definition, \mathcal{D} can be obtained from \mathcal{D}' by removing all bisector edges of \mathcal{B} .

Lemma 3. Each cell of \mathcal{D}' is simply connected.

Proof. Let Q_0 be the decomposition of \mathcal{P} by the edges of \mathcal{B} . Note that Q_0 is exactly $\mathcal{P} \setminus \mathcal{B}$, which is simply connected [27].

Let the points of V be v_1, v_2, \dots, v_{h^*} , ordered arbitrarily. Consider the decomposition Q_1 of Q_0 by the shortest path $\pi(s, v_1)$. Note that Q_1 may have more than one connected cell. Recall that v_1 is on a bisector edge of \mathcal{B} . Since Q_0 is simply connected, $\pi(s, v_1)$ does not cross any bisector edges of $SPM(s)$, and $\pi(s, v_1)$ itself does not form any cycle, each cell of Q_1 is simply connected.

Similarly, consider the decomposition Q_2 of Q_1 by the shortest path $\pi(s, v_2)$. Again, $\pi(s, v_2)$ does not cross any bisector edge of \mathcal{B} . Further, by Observation 1(1), $\pi(s, v_2)$ and $\pi(s, v_1)$ do not cross each other. Hence, $\pi(s, v_2)$ does not cross any edge of Q_1 . Since each cell of Q_1 is simply connected, each cell of Q_2 is also simply connected.

We keep considering the rest of the paths $\pi(s, v_i)$ for $i = 3, 4, \dots, h^*$ one by one in the same way as above. By the similar argument we can obtain that each cell of \mathcal{D}_{h^*} , which is \mathcal{D}' , is simply connected. \square

It is known that $\mathcal{P} \setminus \mathcal{B}$ is simply connected and $\pi(s, t)$ is in $\mathcal{P} \setminus \mathcal{B}$ for any point $t \in \mathcal{P}$ [27]. To simplify the discussion, together with the copies of the points of V , we consider $\mathcal{P}' = \mathcal{P} \setminus \mathcal{B}$ as a simple polygon (with some curved edges) by making two copies for each interior point of every bisector super-curve such that they respectively belong to the two sides of the curve. In this way, for any point $t \in \mathcal{P}'$, it has a unique shortest path $\pi(s, t)$ from s in \mathcal{P}' , which is also a shortest path in \mathcal{P} . In this way, \mathcal{D}' becomes a decomposition of \mathcal{P}' by the tree T_V .

Consider any cell Δ' of \mathcal{D}' . Recall that \mathcal{V} is the set of all vertices of \mathcal{P} . We consider the points of $\mathcal{V} \cup V \cup \{s\}$ on the boundary $\partial\Delta'$ of Δ' as vertices of Δ' . Then, the boundary portion between any two adjacent vertices of Δ' is an obstacle edge, an edge of T_V , or a bisector super-curve. Let p be any point of Δ' . Let $r_{\Delta'}$ be the point of $\Delta' \cap \pi(s, p)$ closest to s . We call $r_{\Delta'}$ the *super-root* of Δ' , which is unique (i.e., independent of p) due to the following lemma.

Lemma 4. 1. The point $r_{\Delta'}$ is in $\mathcal{V} \cup \{s\}$, i.e., it is either s or an obstacle vertex.

2. $\pi(s, r_{\Delta'})$ is a sub-path of a shortest path in Π_V .

3. For any point $t \in \Delta'$, the concatenation of $\pi(s, r_{\Delta'})$ and the shortest path from $r_{\Delta'}$ to t in Δ' is the shortest path $\pi(s, t)$ from s to t in \mathcal{P}' .

Proof. We prove the lemma by induction in a similar way as in Lemma 3. We use the same terminology as in the proof of Lemma 3. Let the points of V be v_1, v_2, \dots, v_{h^*} , ordered arbitrarily.

Let $Q_0 = \mathcal{P} \setminus \mathcal{B}$. For each $1 \leq i \leq h^*$, let Q_i denote the decomposition of Q_{i-1} by $\pi(s, v_i)$. We let $\Pi_0 = \emptyset$. For each $1 \leq i \leq h^*$, let $\Pi_i = \Pi_{i-1} \cup \{\pi(s, v_i)\}$. Hence, $\Pi_V = \Pi_{h^*}$.

Initially, consider the decomposition Q_0 . Note that there is only one cell Δ' in Q_0 . Clearly, $r_{\Delta'} = s$ and all three statements hold for Q_0 and Π_0 . We assume the lemma statements hold for Q_{i-1} and Π_{i-1} . Our goal is to prove that the lemma statements hold for Q_i and Π_i .

Let Δ' be the cell of Q_{i-1} containing v_i . By induction, $\pi(s, v_i)$ is the concatenation of $\pi(s, r_{\Delta'})$ and the shortest path $\pi(r_{\Delta'}, v_i)$ from $r_{\Delta'}$ to v_i in Δ' . Also by induction, $\pi(s, r_{\Delta'})$ is a sub-path of Π_{i-1} . Hence, $\pi(s, v_i)$ does not partition any cell of Q_{i-1} other than Δ' . In other words, for any cell Δ'' of Q_{i-1} , if $\Delta'' \neq \Delta'$, then Δ'' is still in Q_i , and thus the lemma statements still hold on Δ'' and Π_i .

For the cell Δ' , $\pi(r_{\Delta'}, v_i)$ partitions Δ' into multiple *sub-cells*. Consider any sub-cell δ of Δ' . Our goal is to show that the lemma statements hold on δ and Π_i . Depending on whether δ contains $r_{\Delta'}$, there are two cases.

The case $r_{\Delta'} \in \delta$. We first consider the case where δ contains $r_{\Delta'}$. Consider any point p in δ . Since $\delta \subseteq \Delta'$, $r_{\Delta'} \in \delta$, and the point of $\Delta' \cap \pi(s, p)$ closest to s is $r_{\Delta'}$, the point of $\delta \cap \pi(s, p)$ closest to s is also $r_{\Delta'}$. Hence, $r_\delta = r_{\Delta'}$. By induction, the first and second statements of the lemma hold for δ and Π_i .

For the third statement, consider any point $t \in \delta$. Since $t \in \Delta'$, $\pi(s, t)$ is a concatenation of $\pi(s, r_{\Delta'})$ and $\pi(r_{\Delta'}, t)$, and the latter path is in Δ' . To prove the third statement, it sufficient to show that $\pi(r_{\Delta'}, t)$ is in δ . Indeed, assume to the contrary that $\pi(r_{\Delta'}, t)$ is not in δ . Then, since δ is a cell of the decomposition of Δ' by $\pi(r_{\Delta'}, v_i)$, $\pi(r_{\Delta'}, t)$ must cross $\pi(r_{\Delta'}, v_i)$. However, this is not possible due to Observation 1(1). Hence, $\pi(r_{\Delta'}, t)$ must be in δ .

The case $r_{\Delta'} \notin \delta$. Suppose δ does not contain $r_{\Delta'}$. Let a be the point of $\pi(r_{\Delta'}, v_i) \cap \delta$ closest to $r_{\Delta'}$. We first show that for any point $p \in \delta$, a is the point of $\pi(s, p) \cap \delta$ closest to s .

Indeed, since $p \in \Delta'$, $\pi(s, p)$ contains $r_{\Delta'}$ and $\pi(r_{\Delta'}, p)$ is in Δ' . Since $r_{\Delta'}$ is not in δ , let b be the first point in δ we encounter if we traverse on $\pi(r_{\Delta'}, p)$ from $r_{\Delta'}$ to p . Clearly, b is not $r_{\Delta'}$ since otherwise $r_{\Delta'}$ would be in δ . Since δ is a cell of the decomposition of Δ' by $\pi(r_{\Delta'}, v_i)$, b must be on $\pi(r_{\Delta'}, v_i)$. In other words, $b \in \delta \cap \pi(r_{\Delta'}, v_i)$.

Since b is on both $\pi(r_{\Delta'}, v_i)$ and $\pi(r_{\Delta'}, p)$, b is also the first point in δ we encounter if we traverse on $\pi(r_{\Delta'}, v_i)$ from $r_{\Delta'}$ to v_i . Thus, b is the point of $\pi(r_{\Delta'}, v_i)$ closest to $r_{\Delta'}$. Hence, we obtain $b = a$.

On the other hand, the definition of b implies that b is the point of $\pi(s, p) \cap \delta$ closest to s .

Therefore, a is the point of $\pi(s, p) \cap \delta$ closest to s . This implies that $r_\delta = a$.

Note that a is a vertex of $\pi(r_{\Delta'}, v_i)$ and a cannot be v_i . Thus, a must be either s or an obstacle vertex (in fact, a cannot be s either due to $a \neq r_{\Delta'}$), which proves the first statement of the lemma.

Since a is on $\pi(r_{\Delta'}, v_i)$ and thus is on $\pi(s, v_i)$, $\pi(s, a)$ is a sub-path of $\pi(s, v_i) \in \Pi_i$. This proves the second statement of the lemma.

For the third statement, consider any point $t \in \delta$. Since $t \in \Delta'$, by induction, $\pi(s, t)$ is the concatenation of $\pi(s, r_{\Delta'})$ and $\pi(r_{\Delta'}, t)$, and $\pi(r_{\Delta'}, t)$ is in Δ' . Using the same analysis as above, we can show that $\pi(r_{\Delta'}, t)$ must contain a . Further, the portion of $\pi(r_{\Delta'}, t)$ between a and t must be in δ , since otherwise $\pi(r_{\Delta'}, t)$ would cross $\pi(r_{\Delta'}, v_i)$, incurring contradiction. Hence, the portion of $\pi(r_{\Delta'}, t)$ between a and t is the shortest path from a to t in δ . Thus, $\pi(s, t)$ is the concatenation of $\pi(s, a)$ and the shortest path from a to t in δ . This proves the third statement.

This proves that all lemma statements hold for δ and Π_i , and thus hold for Q_i and Π_i .

The lemma thus follows. \square

Observation 3 *Each cell Δ' of \mathcal{D}' has at most one bisector super-curve on its boundary.*

Proof. Assume to the contrary there are two bisector super-curves on the boundary of Δ' . Then, there must exist an endpoint p of one of these two bisector super-curves such that the shortest path $\pi(s, p)$ partitions Δ' into two cells that contain the two bisector super-curves, respectively. This implies that $\pi(s, p)$ is not in Π_V . Since the two endpoints of every bisector super-curve are in V , we obtain $p \in V$ and $\pi(s, p)$ is not in Π_V , a contradiction. \square

Since T_V is a planar tree, we can define its canonical lists as discussed in Section 2. Let v_1 be an arbitrary base leaf of T_V , which can be found in $O(n)$ time. Let the leaf list $\mathcal{L}_l(T_V, v_1)$ be v_1, v_2, \dots, v_{h^*} , which follow the counterclockwise order along $\partial\mathcal{P}'$.

For each $1 \leq i \leq h^*$, let α_i denote the portion of $\partial\mathcal{P}'$ counterclockwise from v_i to v_{i+1} (let v_{h^*+1} refer to v_1). Note that α_i is either a bisector super-curve or a chain of obstacle edges. Suppose we move a point t on α_i from v_i to v_{i+1} . The shortest path $\pi(s, t)$ will continuously change with the same topology since $\pi(s, t)$ is always in \mathcal{P}' (which is simply connected). Let R_i be the region of \mathcal{P}' that is “swept” by $\pi(s, t)$ during the above movement of t . More specifically, let p_i be the common point on $\pi(s, v_i) \cap \pi(s, v_{i+1})$ that is farthest to s . Then, R_i is bounded by $\pi(p_i, v_i)$, $\pi(p_i, v_{i+1})$, and α_i . For convenience of discussion, we let R_i also contain the common sub-path $\pi(s, p_i) = \pi(s, v_i) \cap \pi(s, v_{i+1})$ and we call $\pi(s, p_i)$ the *tail* of R_i . We call the region bounded by $\pi(p_i, v_i)$, $\pi(p_i, v_{i+1})$, and α_i the *cell* of R_i . We consider $\pi(s, v_i)$, $\pi(s, v_{i+1})$, and α_i as the three portions of the boundary ∂R_i of R_i . The definition implies that for any point t in R_i , $\pi(s, t)$ is in R_i . In fact, if t is in the cell of R_i , then $\pi(s, t)$ is the concatenation of $\pi(s, p_i)$ and the shortest path from p_i to t in the cell. Clearly, \mathcal{P}' is the union of R_1, R_2, \dots, R_{h^*} . Let $\mathcal{R} = \{R_1, R_2, \dots, R_{h^*}\}$. The next lemma is proved with the help of the regions of \mathcal{R} . The set \mathcal{R} will also be quite useful in Section 4. Recall that each edge of $\partial\Delta'$ is either an obstacle edge, a bisector super-curve, or an edge of T_V (also called a *shortest path edge*).

Lemma 5. *For each cell Δ' of \mathcal{D}' , there are two shortest paths of Π_V that contain all shortest path edges of $\partial\Delta'$.*

Proof. By the definitions of the regions of \mathcal{R} , Δ' is contained in the cell of a region R_i of \mathcal{R} . Therefore, each shortest path edge of $\partial\mathcal{D}'$ belongs to either $\pi(s, v_i)$ or $\pi(s, v_{i+1})$. \square

Observe that the decomposition \mathcal{D} can be obtain from \mathcal{D}' by removing all bisector super-curves. For any bisector super-curve α , the two cells of \mathcal{D}' incident to α are merged into one cell of \mathcal{D} . Due to Observation 3, a cell of \mathcal{D}' can be merged into at most one cell of \mathcal{D} . Therefore, for each cell Δ of \mathcal{D} , either Δ is also in \mathcal{D}' or Δ is a *merged cell* merged by exactly two cells of \mathcal{D}' . Since every cell of \mathcal{D}' is simply connected, each cell of \mathcal{D} is also simply connected. This proves Lemma 1(3).

Consider any line segment $\tau \in \mathcal{P}$. By Observation 1(2), τ can cross any shortest path of Π_V at most once. Hence, τ can cross the shortest paths of Π_V at most $O(h)$ times in total. Whenever τ crosses the boundary of a cell of \mathcal{D} , it must cross a shortest path of Π_V . Thus, τ can intersect $O(h)$ cells of \mathcal{D} . This proves the first part of Lemma 1(4). For the second part, consider any cell Δ . By Lemma 5, if Δ is not a merged cell, then τ can cross the boundary of Δ at most twice; otherwise, τ can cross the boundary of Δ at most four times. Therefore, the intersection $\tau \cap \Delta$ consists of at most two (maximal) sub-segments of τ . This proves the second part of Lemma 1(4).

In the sequel, we prove Lemma 1(5). Consider any cell Δ of \mathcal{D} . According to our discussion above, Δ is either in \mathcal{D}' or a merged cell of two cells Δ_1 and Δ_2 of \mathcal{D}' . If it is the former case,

then we also call r_Δ the *super-root* of Δ ; otherwise, we call r_{Δ_1} and r_{Δ_2} the two *super-roots* of Δ . Lemma 4 leads to the following lemma, which proves Lemma 1(6).

Lemma 6. *For any cell Δ of \mathcal{D} , the following hold.*

1. *Its two super-roots are in $\mathcal{V} \cup \{s\}$.*
2. *For each super-root r of Δ , $\pi(s, r)$ is a sub-path of a shortest path in Π_V .*
3. *For any point $t \in \Delta$, $\pi(s, t)$ is the concatenation of $\pi(s, r)$ and the shortest path from r to t in Δ , for a super-root r of Δ .*

Proof. By Lemma 4, the proof is straightforward because either Δ is a cell of \mathcal{D} or a merge of two cells of \mathcal{D} . \square

Recall that for any simple polygon P and a fixed source point, each segment query can be answered in $O(\log |P|)$ time after $O(|P|)$ time preprocessing [1]. As preprocessing, for each cell Δ of \mathcal{D} , since it is a simple polygon, we compute the above segment query data structure with respect to each super-root of Δ . This takes $O(n)$ time and space in total by Lemma 1(2).

Consider any segment τ' in a cell Δ of \mathcal{D} . By Lemma 6, $\pi(s, \tau')$ is the concatenation of $\pi(s, r)$ from s to a super-root r of Δ and the shortest path $\pi(r, \tau')$ from r to τ' in Δ . As r is in $\mathcal{V} \cup \{s\}$ by Lemma 6(1), $\pi(s, r)$ is available from $SPM(s)$, and $\pi(r, \tau')$ can be found in $O(\log |\Delta|)$ time. Hence, our query algorithm works as follows. For each super-root r of Δ , we compute $\pi(s, r)$ and $\pi(r, \tau')$ to obtain a “candidate” shortest path from s to τ' . Then, we return the shorter one of the at most two candidates paths as the solution. The total time is $O(\log |\Delta|)$. This proves Lemma 1(5).

Remark. One may wonder why we do not use \mathcal{D}' instead of \mathcal{D} to answer the segment queries. The reason is that the boundaries of cells of \mathcal{D}' contain bisector super-curves and the query segment τ may intersect a bisector super-curve multiple times, and thus a similar observation as Lemma 1(4) cannot be guaranteed on \mathcal{D}' .

Finally, we prove Lemma 1(7) in the following lemma.

Lemma 7. *Given $SPM(s)$, the decomposition \mathcal{D} can be computed in $O(n)$ time.*

Proof. Let \mathcal{D}_1 be the decomposition of $SPM(s)$ by the edges of $SPT(s)$. As discussed before, we can easily obtain $SPT(s)$ from $SPM(s)$ and thus obtain \mathcal{D}_1 in $O(n)$ time. Further, for each point $v \in V$, we add to \mathcal{D}_1 the last edge of the shortest path $\pi(s, v)$, which is also the edge connecting v to the root of the cell of $SPM(s)$ containing v . Let \mathcal{D}_2 be the resulting decomposition, which can be obtained in $O(n)$ time. Note that each edge of T_V is also an edge of \mathcal{D}_2 .

Since \mathcal{D} is a decomposition of \mathcal{P} by the edges of T_V , \mathcal{D} can be obtained from \mathcal{D}_2 by removing those edges that are not in \mathcal{D} . To this end, we first remove all bisector edges from \mathcal{D}_2 . Then, we remove the edges of $SPT(s)$ that are not in T_V . This can be done by first marking all edges of T_V in \mathcal{D}_2 and then removing all unmarked edges of $SPT(s)$ from \mathcal{D}_2 . Below we only discuss how to mark all edges of T_V in $O(n)$ time since the latter step is trivial.

For each vertex v of V , we mark the edges of $\pi(s, v)$ in \mathcal{D}_2 as follows. We start from v and traverse along $\pi(s, v)$ from v to s , marking every edge that has not been marked yet; we stop the traversal either when we encounter s or we encounter an edge that has been marked. In this way, every edge of T_V is marked exactly once. Since T_V has $O(n)$ edges, the above marking algorithm runs in $O(n)$ time.

Thus, the decomposition \mathcal{D} can be computed in $O(n)$ time. \square

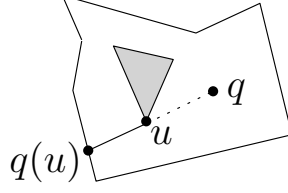


Fig. 3. Illustrating a window $\overline{uq(u)}$ of q .

4 The Quickest Visibility Queries: The Preliminary Result

In this section, we give our preliminary result on quickest visibility queries, which sets the stage for our improved result in Section 5.

For any subset A of \mathcal{P} , a point $p \in A$ is called a *closest point* of A (with respect to s) if $d(s, A) = d(s, p)$.

Given any query point q in \mathcal{P} , our goal is to find a shortest path from s to $\text{Vis}(q)$. Let q^* be a closest point of $\text{Vis}(q)$. To answer the query, it is sufficient to determine q^* . Thus we will focus on finding q^* . Note that if q is visible to s , then $q^* = s$. We can determine whether s is visible to q in $O(\log n)$ time by checking whether q is in the cell of $\text{SPM}(s)$ whose root is s . In the following, we assume s is not visible to q .

We define the *windows* of q and $\text{Vis}(q)$, which were used for studying the visibility polygons, e.g., [5,10]. Consider an obstacle vertex u that is visible to q such that the two incident obstacle edges of u are on the same side of the line through q and u (e.g., see Fig. 3). Let $q(u)$ denote the first point on $\partial\mathcal{P}$ hit by the ray from u along the direction from q to u . Then $\overline{uq(u)}$ is called a *window* of q ; we say that the window is *defined by* u . Further, we call $\overline{qq(u)}$ the *extended window* of $uq(u)$.

Each window of q is an edge of $\text{Vis}(q)$, and thus the number of windows of q is $O(K)$, where $K = |\text{Vis}(q)|$. Further, there must be a closest point q^* that is on a window of q [1]. Hence, as in [1], a straightforward algorithm to compute q^* is to compute shortest paths from s to all windows of s and the path of minimum length determines q^* . To compute shortest paths from s to all windows, if we apply our segment queries on all windows using Theorem 1, then the total time would be $O(K \cdot h \cdot \log \frac{n}{h})$. In the rest of this section, we present an algorithm that can compute q^* in $O((K + h) \log h \log n)$ time, without having to compute shortest paths to all windows. The key idea is to prune some (portions of) windows such that q^* is still in the remaining windows and the shortest paths from s to all remaining windows can be computed efficiently.

4.1 The Algorithm Overview

As the first step, we compute $\text{Vis}(q)$, which can be done in $O(K \log n)$ time after $O(n + h^2 \log h)$ time and $O(n + h^2)$ space preprocessing [9]. Then, we can find all windows and extended-windows in $O(K)$ time. For ease of exposition, we make a general position assumption for q that q is not collinear with any two obstacle vertices. The assumption implies that q is in the interior of \mathcal{P} and no two windows are collinear.

Let u_0 be the root of the cell of $\text{SPM}(s)$ containing q (if q is on the boundary of multiple cells, then we take an arbitrary such cell). Hence, $\pi(s, u_0) \cup \overline{u_0q}$ is a shortest path $\pi(s, q)$ from s to q . Note that u_0 must define a window $\overline{u_0q(u_0)}$ of q [27]. Let $\overline{u_0q(u_0)}, \overline{u_1q(u_1)}, \dots, \overline{u_kq(u_k)}$ be all windows of q ordered *clockwise* around q . Clearly, $k = O(K)$. For each $0 \leq i \leq k$, let $q_i = q(u_i)$.

Note that the window $\overline{u_0q_0}$ is special in the sense that u_0 is in $\pi(s, q)$. So we first apply our algorithm in Theorem 1 on $\overline{u_0q_0}$ to compute the closest point q_0^* of $\overline{u_0q_0}$. Clearly, if $q^* \in \overline{u_0q_0}$, then $q^* = q_0^*$. In the following, we assume $q^* \notin \overline{u_0q_0}$. Let $Q = \{q, q_1, q_2, \dots, q_k\}$. Note that Q does not contain q_0 but q . If $q^* \in Q$, then we can find q^* by computing $d(s, p)$ for all $p \in Q$, which can be done in $O(k \log n)$ time using $SPM(s)$. In the following, we assume $q^* \notin Q$. Note that the above assumption that $q^* \notin \overline{u_0q_0} \cup Q$ is only for arguing the correctness of our following algorithm, which actually proceeds without knowing whether the assumption is true or not.

For each $0 \leq i \leq k$, let $w_i = \overline{qq_i}$, i.e., the extended window of $\overline{u_iq_i}$. Let $W = \{w_i \mid 1 \leq i \leq k\}$. For convenience of discussion, we assume that each w_i of W does not contain its two endpoints q and q_i (but the endpoints of w_i still refer to q and q_i). Since $q^* \notin \overline{u_0q_0} \cup Q$, q^* must be on an extended window of W . Clearly, q^* is also a closest point of W . Since no two windows of q are collinear, no extended-window of W contains another. We assign each window $w_i \in W$ a direction from q to q_i , so that we can talk about its left or right side.

Suppose q^* is on $w_i \in W$. Since w_i is an open segment, by the definition of q^* , the shortest path $\pi(s, q^*)$ must reach q^* from either the left side or the right side of w_i . Formally, we say that $\pi(s, q^*)$ reaches q^* from the left side (resp., right side) of w_i if there is a small neighborhood of q^* such that all points of $\pi(s, q^*)$ in the neighborhood are on the left side (resp., right side) of w_i . Let w_i^l (resp., w_i^r) denote the set of points p on w_i whose shortest path from s to p is from the left (resp., right) side of w_i . Hence, q^* is either on w_i^l or on w_i^r .

Our algorithm will find two points q_l^* and q_r^* such that if q^* is on w_i^l for some $i \in [1, k]$, then $q^* = q_l^*$, and otherwise (i.e., q^* is in w_i^r for some $i \in [1, k]$), $q^* = q_r^*$.

In the following, we will only present our algorithm for finding q_l^* since the case for q_r^* is symmetric. In the following discussion, we assume q^* is on w_i^l for some $i \in [1, k]$. Note that this assumption is only for arguing the correctness of our algorithm, which actually proceeds without knowing whether the assumption is true.

The rest of this section is organized as follows. In Section 4.2, we discuss some observations, based on which we describe our pruning algorithm in Section 4.3 to prune some (portions of) segments of W such that q^* ($= q_l^*$) is still in the remaining segments of W . In Section 4.5, we will finally compute q_l^* (which will be q^*) on the remaining segments of W . Some implementation details of the algorithm are given in Sections 4.4 and 4.6. Section 4.7 summarizes the overall algorithm.

As will be clear later, our algorithm uses extended windows instead of windows because extended windows can help us with the pruning.

4.2 Observations

For any point $t \in \mathcal{P}$ with $s \neq t$, and its shortest path $\pi(s, t)$, we use t^+ to denote a point on $\pi(s, t)$ infinitely close to t (but $t^+ \neq t$). If t is on w_i^l for some $i \in [1, k]$, then t^+ must be on the left side of w_i .

For any segment w of W , we say that w or a sub-segment of w can be *pruned* if it does not contain q^* . Our pruning algorithm, albeit somewhat involved, is based on the following simple observation.

Observation 4 *For any point $t \in w_i^l$ for some $i \in [1, k]$, if $\pi(s, t^+)$ intersects any segment $w \in W$ or an endpoint of it, then t can be pruned (i.e., t cannot be q^*).*

Proof. Let t' be a point on $\pi(s, t^+)$ that is a point on any segment $w \in W$ or an endpoint of it. Clearly, $t' \in Vis(s)$ and $d(s, t') < d(s, t)$. Thus, t cannot be q^* . \square

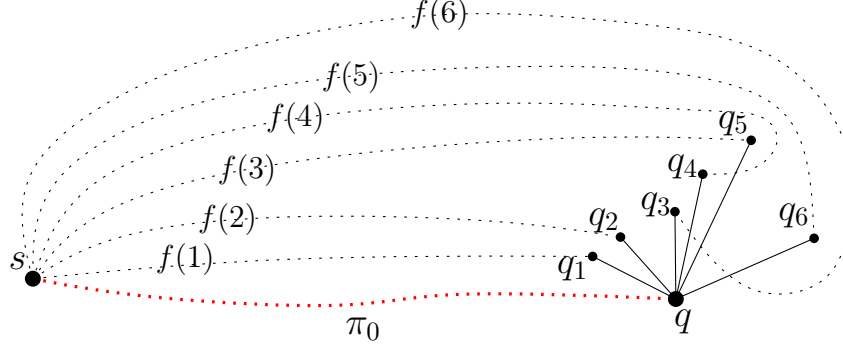


Fig. 4. Illustrating the map $f(\cdot)$: $f(1) = 1$, $f(2) = 2$, $f(3) = 5$, $f(4) = 4$, $f(5) = 6$, and $f(6) = 3$. Note that the paths could be “below” π_0 , but for ease of exposition, we “flip” them above π_0 , and this flip operation does not change the topology of these paths.

Consider the shortest paths $\pi(s, q_i)$ for $i = 1, 2, \dots, k$. To simplify the notation, let $\pi_i = \pi(s, q_i)$ for each $i \in [1, k]$. In particular, let $\pi_0 = \pi(s, q)$ (not $\pi(s, q_0)$). Recall that $Q = \{q, q_1, \dots, q_k\}$. The union of all paths π_i for $0 \leq i \leq k$ forms a planar tree, denoted by T_Q , with root at s . Consider the canonical cycle $\mathcal{C}(T_Q)$ as defined in Section 2. Let \mathcal{C}_Q be the circular list of the points of Q following their relative order in $\mathcal{C}(T_Q)$. We further break \mathcal{C}_Q into a list \mathcal{L}_Q at q , such that \mathcal{L}_Q starts from q and all other points of \mathcal{L}_Q follow the counterclockwise order in \mathcal{C}_Q . Assume \mathcal{L}_Q is $\{q, q_{f(1)}, q_{f(2)}, \dots, q_{f(k)}\}$, i.e., the $(i+1)$ -th point of the list is $q_{f(i)}$; e.g., see Fig. 4. So $f(\cdot)$ essentially maps each point of $Q \setminus \{q\}$ from its position in \mathcal{L}_Q to its position in the list $\{q_1, q_2, \dots, q_k\}$. Hence, $f(1) \dots, f(k)$ is a permutation of $1, \dots, k$, and $f(i) \neq f(j)$ if $i \neq j$. The reason we introduce the list \mathcal{L}_Q is that intuitively, for any $1 \leq i < j \leq k$, the path $\pi_{f(j)}$ is *counterclockwise* from $\pi_{f(i)}$ with respect to π_0 around s . For convenience, we let $f(0) = 0$.

Later in Section 4.6 we will give the implementation details for the following lemma.

Lemma 8. *Given $SPM(s)$, after $O(n)$ time preprocessing, we can compute the list \mathcal{L}_Q and thus determine the map $f(\cdot)$ in $O(k \log n)$ time.*

Observation 5 *For any $i \in [1, k]$, π_0 does not contain q_i and π_i does not contain q .*

Proof. Assume to the contrary that π_0 contains q_i for some $i \in [1, k]$. Since q is in π_0 , by Observation 1(2), $\pi_0 = \pi_i \cup \overline{q_i q}$. Recall that $\overline{q q_0} \in \pi_0$. This implies that either $\overline{q q_0}$ contains q_i or $\overline{q q_i}$ contains q_0 , which further implies the two windows $\overline{u_0 q_0}$ and $\overline{u_i q_i}$ are collinear. This incurs contradiction since no two windows are collinear. Hence, π_0 does not contain q_i .

Assume to the contrary that π_i contains q . Then, since both q and q_i are in π_i , by Observation 1(2), $\overline{q q_i}$ is in π_i . Hence, $\pi_i = \pi_0 \cup \overline{q q_i}$. Recall that u_0 is the root of the cell of $SPM(s)$ containing q , and $\pi_0 = \pi(s, u_0) \cup \overline{u_0 q}$. Since q is in the interior of \mathcal{P} , $\overline{u_0 q}$ and $\overline{q q_i}$ must be collinear since otherwise there would be a shorter path from u_0 to q_i without containing $\overline{q q_i}$. Recall that $u_i \in \overline{q q_i}$. Since $\overline{u_0 q}$ and $\overline{q q_i}$ are collinear, the three points q , u_0 , and u_i are collinear. But this contradicts with our general position assumption that q is not collinear with any two obstacle vertices. \square

Lemma 9. *Suppose π_j contains q_i with $i \neq j$ and $i, j \in [1, k]$. If $i < j$, then w_j can be pruned; otherwise, w_i can be pruned.*

Proof. We first discuss the case $i < j$. Consider the region D bounded by the closed curve that is the union of w_i , w_j , and the subpath of π_j between q_i and q_j (e.g., see Fig. 5(a)). By Observation 1(1),

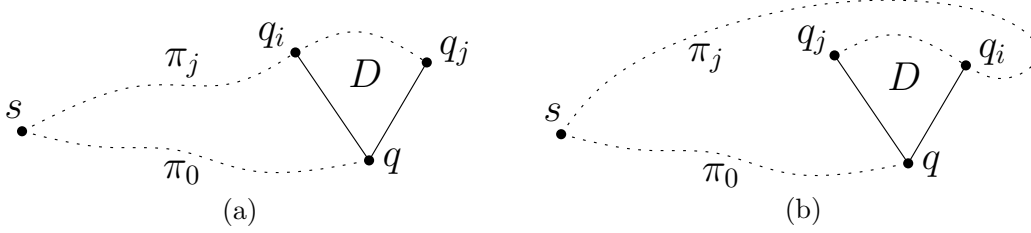


Fig. 5. Illustrating the proof of Lemma 9: (a) the case $i < j$; (b) the case $i > j$.

π_j does not cross π_0 . Since $i < j$, w_j is clockwise from w_i with respect to w_0 (which is the last edge of π_0). Hence, D must be locally on the left side of w_j .

Consider any point $t \in w_j^l$. We show that t cannot be q^* . Recall that w_j is an open segment, so t is not q or q_j . Since $t \in w_j^l$, the point t^+ must be in D . By the definition of D , s is not in the interior of D . Hence, $\pi(s, t^+)$ must intersect the boundary of D . Since $\pi(s, t^+)$ cannot cross the subpath of π_j between q_i and q_j , $\pi(s, t^+)$ must intersect w_i , w_j , or a point of $\{q, q_i, q_j\}$. By Observation 4, t cannot be q^* .

The above shows that t cannot be q^* . Thus, w_j can be pruned.

For the case $i > j$, the argument is similar (e.g., see Fig. 5(b)). Since $i > j$, D must be locally on the left side of w_i . For any point $t \in w_i^l$, using the similar argument as above, we can show that t cannot be q^* . Thus, w_i can be pruned. \square

Lemma 10 provides an algorithm to remove all extended-windows of W that can be pruned by Lemma 9.

Lemma 10. *Given $SPM(s)$ and with $O(n)$ time preprocessing, we can find in $O(k \log n)$ time all segments of W that can be pruned by Lemma 9.*

Proof. The task is to determine those indices i and j such that q_i is contained in π_j for $i \neq j$ in $[1, k]$, after which we can determine whether w_i or w_j should be pruned by Lemma 9. Recall that $f(1), f(2), \dots, f(k)$ is a permutation of the indices of $\{1, 2, \dots, k\}$. Therefore, equivalently we can determine those indices $f(i)$ and $f(j)$ such that $q_{f(i)}$ is contained in $\pi_{f(j)}$ for $f(i) \neq f(j)$ in $[1, k]$. We actually do not need to explicitly find all such pairs, as shown below.

A key observation is that if $q_{f(i)}$ is contained in a path $\pi_{f(j)}$ with $f(j) \neq f(i)$, then it must be that $j < i$ and $q_{f(i)}$ is contained in $\pi_{f(m)}$ for any $m \in [j, i]$. Indeed, if $q_{f(i)}$ is contained in a path $\pi_{f(j)}$ with $f(j) \neq f(i)$, then the subpath of $\pi_{f(j)}$ from s to $q_{f(i)}$ is $\pi_{f(i)}$. According to the definition of the map $f(\cdot)$, i.e., the list $\{q_{f(1)}, q_{f(2)}, \dots, q_{f(k)}\}$, $q_{f(i)}$ must be after $q_{f(j)}$ in the list, i.e., $j < i$. Further, for any $m \in [j, i]$, $q_{f(i)}$ is in the path of the tree T_Q from $q_{f(m)}$ to the root s , which is the shortest path $\pi_{f(m)}$.

Based on the above observation, our algorithm works as follows. We consider the points $q_{f(i)}$ in the order of $i = 1, 2, \dots, k$. Suppose we are about to process $q_{f(i)}$. The algorithm maintains a stack S of indices in $[1, i-1]$ in increasing order (from bottom to top of S) such that for each $j \in [1, i-1]$, if $j \notin S$, then $w_{f(j)}$ has been pruned. Initially we set $S = \emptyset$ before we process $q_{f(1)}$. In general, our algorithm processes $q_{f(i)}$ for any $i \geq 1$ as follows.

If $S = \emptyset$, then we push i on top of S and proceed to process $q_{f(i+1)}$. Otherwise, we first check whether $q_{f(i)}$ is contained in $\pi_{f(m)}$, where m is the top index on S .

1. If $q_{f(i)} \notin \pi_{f(m)}$, then $q_{f(i)}$ is not in any path $\pi_{f(j)}$ with $j < m$ by the above observation. We push i on top of S and then proceed on processing $q_{f(i+1)}$.

2. If $q_{f(i)} \in \pi_{f(m)}$, then depending on whether $f(i) < f(m)$, there are two cases.
 - (a) If $f(i) < f(m)$, then by Lemma 9, we prune $w_{f(m)}$ and pop m from S . Then, we repeat the same algorithm as above (i.e., first check whether $S = \emptyset$, and if not, check whether $q_{f(i)}$ is contained in $\pi_{f(m)}$, where m is the new top index of S).
 - (b) If $f(i) > f(m)$, then by Lemma 9, we prune $w_{f(i)}$ and proceed on processing $q_{f(i+1)}$.

The algorithm finishes once $q_{f(k)}$ has been processed. It is not difficult to see that if we can check whether $q_{f(i)}$ is in $\pi_{f(m)}$ in $O(c)$ time, then the algorithm runs in $O(k \cdot c)$ time since each index of $[1, k]$ can be pushed or popped from S at most once. In the following, we show that $c = O(\log n)$ after $O(n)$ time preprocessing, and this will prove the lemma.

First of all, if both $q_{f(i)}$ and $q_{f(m)}$ are in the same cell σ of $SPM(s)$, then $q_{f(i)} \in \pi_{f(m)}$ if and only if $q_{f(i)} \in r(\sigma)q_{f(m)}$, where $r(\sigma)$ is the root of σ . Otherwise, if $q_{f(i)}$ is not in any edge of the shortest path tree $SPT(s)$, then $q_{f(i)}$ cannot be in $\pi_{f(m)}$. Otherwise, suppose $q_{f(i)}$ is on an edge e of $SPT(s)$. We can find the edge e in $O(\log n)$ time by a point location query on the decomposition of $SPM(s)$ by the edges of $SPT(s)$. Let v be an endpoint of e , and thus v is a node of $SPT(s)$. Let r be the root of the cell of $SPM(s)$ containing $q_{f(m)}$. Then, $q_{f(i)}$ is in $\pi_{f(m)}$ if and only if v is an ancestor of r in $SPT(s)$. Note that v is an ancestor of r if and only if the lowest common ancestor of v and r is v . We can build a data structure on $SPT(s)$ in $O(n)$ time such that given any two nodes of the tree, the lowest common ancestor can be found in constant time [3,18].

Hence, we can determine whether $q_{f(i)} \in \pi_{f(m)}$ in $O(\log n)$ time after $O(n)$ time preprocessing. The lemma thus follows. \square

We apply the algorithm in Lemma 10 to prune the segments of W . But to simplify the notation, we assume that none of the segments of W is pruned since otherwise we could re-index all segments of W . So now W has the following property.

Observation 6 *For any $i \in [1, k]$, q_i is not contained in any π_j with $j \in [0, k]$ and $j \neq i$.*

Proof. Suppose to the contrary that q_i is contained in π_j for some $j \in [0, k]$ and $i \neq j$. On the one hand, due to Observation 5, $j \neq 0$. On the other hand, if $j \in [1, k]$, then by Lemma 9 either w_i or w_j would have already been removed from W . \square

For each $i \in [1, k]$, since π_0 does not cross π_i , $\pi_0 \cup \pi_i \cup w_i$ forms a closed curve that separates the plane into two regions, one locally on the left of w_i and the other locally on the right w_i . We let D_i denote the region locally on the left side of w_i including $\pi_0 \cup \pi_i \cup w_i$ as its boundary (it is possible that D_i is unbounded). If $\pi_0 \cap \pi_i$ is a sub-path including at least one edge, then it is also considered to be in D_i . We have the following observation for D_i .

Observation 7 *If $q^* \in w_i^l$, then $\pi(s, q^*)$ must be in D_i .*

Proof. Let $t = q^*$ that is on w_i^l . Then, t^+ is in the interior of D_i . By Observation 4, $\pi(s, t^+)$ cannot intersect w_i . Also, $\pi(s, t^+)$ cannot cross either π_0 or π_i , and s is on the boundary of D_i . Hence, $\pi(s, t^+)$ must be inside D_i . Thus, $\pi(s, q^*)$ is in D_i . \square

Our pruning algorithm mainly relies on the following lemma, whose proof in turn boils down to Observation 4.

Lemma 11. *Suppose i and j are two indices with $1 \leq i < j \leq k$.*

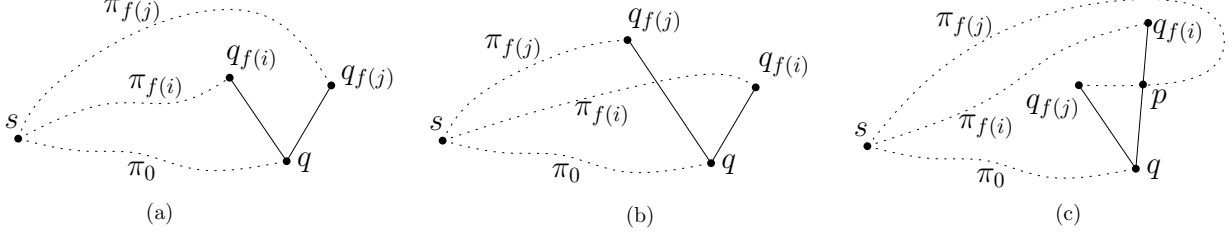


Fig. 6. Illustrating Lemma 11.

1. If $f(i) < f(j)$, then $\pi_{f(i)}$ does not cross $w_{f(j)}$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, and further, $D_{f(i)}$ is contained in $D_{f(j)}$ (e.g., see Fig. 6(a)).
2. If $f(i) > f(j)$, then either $\pi_{f(i)}$ crosses $w_{f(j)}$ or $\pi_{f(j)}$ crosses $w_{f(i)}$. Further, in the former case (e.g., see Fig. 6(b)), $w_{f(i)}$ can be pruned, and in the latter case (e.g., see Fig. 6(c)), the sub-segment \overline{qp} of $w_{f(i)}$ can be pruned, where p is the point at which $\pi_{f(j)}$ crosses $w_{f(i)}$.

Proof. Suppose $f(i) < f(j)$. We first show that $q_{f(j)}$ cannot be in the interior of the region $D_{f(i)}$.

Assume to the contrary that $q_{f(j)}$ is in the interior of $D_{f(i)}$. Let $p_{f(j)}$ be a point on $w_{f(j)}$ arbitrarily close to q (but $p_{f(j)} \neq q$). Since $f(i) < f(j)$, $w_{f(j)}$ is clockwise from $w_{f(i)}$ with respect to w_0 . Since q is not in $\pi_{f(i)}$ by Observation 5, $p_{f(j)}$ is not in $D_{f(i)}$. Since $q_{f(j)}$ is in the interior of $D_{f(i)}$, $\pi_{f(i)}$ must cross $w_{f(j)}$ at a point p with $p \neq q_{f(j)}$ (e.g., see Fig. 7).

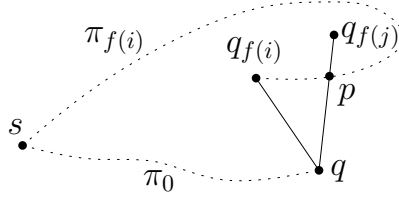


Fig. 7. Illustrating the scenario where $q_{f(j)}$ is in the interior of $D_{f(i)}$.

Depending on whether $p \in \pi_{f(j)}$, there are two cases.

1. If $p \in \pi_{f(j)}$, then since $p \in w_{f(j)}$, we obtain $\pi_{f(j)} = \pi(s, p) \cup \overline{pq_{f(i)}}$ by Observation 1(2). Since $q_{f(j)}$ is in the interior of $D_{f(i)}$, we further obtain that $\pi_{f(i)}$ is counterclockwise from $\pi_{f(j)}$ with respect to π_0 . Thus, we have $i > j$, a contradiction.
2. If $p \notin \pi_{f(j)}$, then since $i < j$ and $\pi_{f(j)}$ is counterclockwise from $\pi_{f(i)}$ with respect to π_0 , $\pi_{f(j)}$ must cross an interior point p' of \overline{qp} before reaching $q_{f(j)}$. This implies that $\pi_{f(j)} = \pi(s, p') \cup \overline{p'q_{f(j)}}$ by Observation 1(2), and thus, $\pi_{f(j)}$ contains p since $p \in \overline{p'q_{f(i)}}$. Hence, we again obtain contradiction.

This proves that $q_{f(j)}$ cannot be in the interior of the region $D_{f(i)}$.

By Observations 5 and 6, $q_{f(j)}$ cannot be in π_0 or $\pi_{f(i)}$. Since no segment of W contains another, $q_{f(j)}$ cannot be in $w_{f(i)}$. Hence, $q_{f(j)}$ cannot be on the boundary of $D_{f(i)}$. Therefore, $q_{f(j)}$ is outside $D_{f(i)}$. Next we show that $\pi_{f(i)}$ does not cross $w_{f(j)}$.

Indeed, since both $q_{f(j)}$ and $p_{f(j)}$ are outside $D_{f(i)}$, in order for $\pi_{f(i)}$ to cross $w_{f(j)}$, $\pi_{f(i)}$ must cross $w_{f(j)}$ at least twice, which is not possible by Observation 1(2). Similarly, in order for $\pi_{f(j)}$ to cross $w_{f(i)}$, it would have to cross $w_{f(i)}$ at least twice, which is not possible.

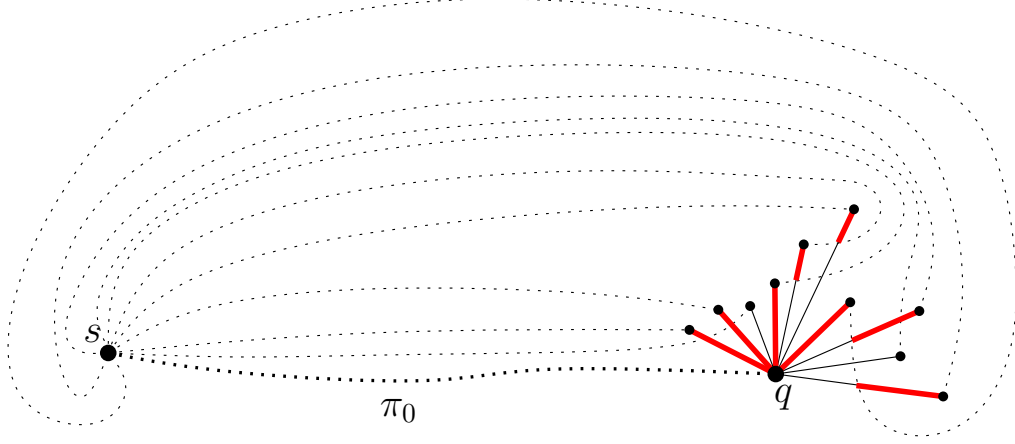


Fig. 8. The thick (red) segments are the remaining parts of the segments of W after the pruning algorithms (so that q_i^* must be on the left side of a red segment). Note that the paths could be “below” π_0 , but for ease of exposition, we “flip” them above π_0 , and this flip operation actually does not change the topology of these paths.

This proves that $\pi_{f(i)}$ does not cross $w_{f(j)}$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$. Since $w_{f(j)}$ is clockwise from $w_{f(i)}$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, $w_{f(i)}$ is contained in $D_{f(j)}$. Further, since $\pi_{f(i)}$ is counterclockwise from $\pi_{f(j)}$ and $\pi_{f(i)}$ does not cross $w_{f(j)}$, $D_{f(i)}$ must be contained in $D_{f(j)}$.

This proves the first part of the lemma.

For the second part of the lemma, we assume $f(i) > f(j)$. By the same analysis as above, $q_{f(i)}$ cannot be on the boundary of $D_{f(j)}$. Depending on whether $q_{f(i)}$ is in the interior of $D_{f(j)}$ or outside it, there are two cases.

1. If $q_{f(i)}$ is outside $D_{f(j)}$, then since $i < j$, $\pi_{f(j)}$ is counterclockwise from $\pi_{f(i)}$ with respect to π_0 . Further, since $\pi_{f(i)}$ and $\pi_{f(j)}$ do not cross each other and $\pi_{f(i)}$ does not contain q (by Observation 5), $\pi_{f(i)}$ must cross $w_{f(j)}$. Let p be the point of $w_{f(j)}$ where $\pi_{f(i)}$ crosses. Let D be the open region bounded by $w_{f(i)}$, \overline{qp} , and the subpath π' of $\pi_{f(i)}$ between p and $q_{f(i)}$. Consider any point t on $w_{f(i)}^l$ (if any). The point t^+ must be in the interior of D . Clearly, s is not in D . Hence, $\pi(s, t^+)$ must cross the boundary of D . Since $\pi(s, t^+)$ cannot cross π' , it must cross either \overline{qp} or $w_{f(i)}$. By Observation 4, t can be pruned. Thus, $w_{f(i)}$ can be pruned.
2. If $q_{f(i)}$ is in the interior of $D_{f(j)}$, let $p_{f(i)}$ be a point on $w_{f(i)}$ infinitely close to q . Since $f(i) > f(j)$, by the same analysis as before, $p_{f(i)}$ is not in $D_{f(j)}$. Since $q_{f(i)}$ is in the interior of $D_{f(j)}$, $\overline{q_{f(i)}p_{f(i)}}$ must intersect the boundary of $D_{f(j)}$ at a point p . Since $\overline{q_{f(i)}p_{f(i)}}$ does not intersect π_0 or $w_{f(j)}$, p is on $\pi_{f(j)}$. This proves that $\pi_{f(j)}$ crosses $w_{f(i)}$. Consider the region D bounded by \overline{qp} , $w_{f(j)}$, and the subpath of $\pi_{f(j)}$ between p and $q_{f(j)}$. Consider any point t on $\overline{qp} \cap w_{f(i)}^l$. By the similar argument as above, we can show that t can be pruned. Thus, \overline{qp} can be pruned.

The lemma thus follows. □

For any $1 \leq i < j \leq k$, we say π_i and π_j are *consistent* if $f(i) < f(j)$. By Lemma 11, if π_i and π_j are not consistent, then we can do some pruning, based on which we present our pruning algorithm in Section 4.3. Figure 8 gives an example showing the remaining parts of the segments of W after the pruning algorithm.

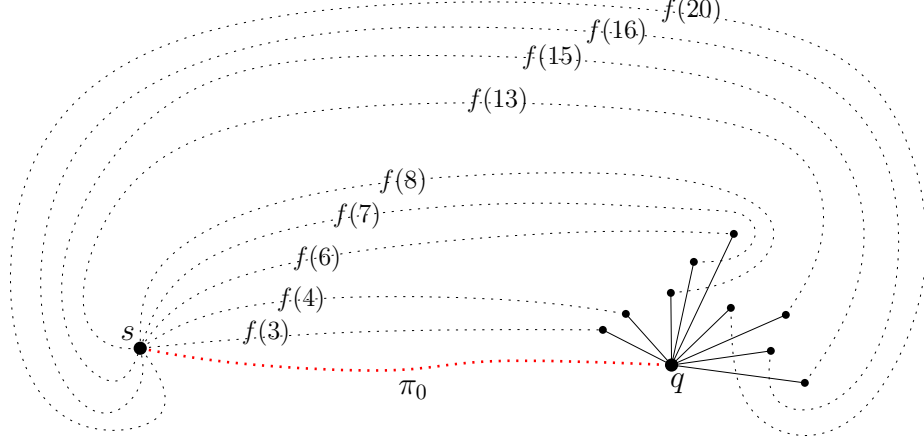


Fig. 9. Illustrating the shortest paths corresponding to the indices in the current bundle sequence $\mathbb{B} = \{\{3\}, \{4\}, \{\{6\}, \{7\}\}, \{8\}\}, \{\{13\}, \{\{15\}, \{16\}\}, \{20\}\}$, where each underline indicates a bundle of \mathbb{B} . For example, the last bundle is a composite bundle consisting of three children bundles with 20 as its wrap index. In the figure, the indices of the paths are labeled. Note that the paths could be “below” π_0 , but for ease of exposition, we “flip” them above π_0 , and this flip operation actually does not change the topology of these paths.

4.3 A Pruning Algorithm for Pruning the Segments of W

We process the paths $\pi_{f(1)}, \pi_{f(2)}, \dots, \pi_{f(k)}$ in this order. Assume that $\pi_{f(i-1)}$ has just been processed and we are about to process $\pi_{f(i)}$. Our algorithm maintains a sequence of *bundles*, denoted by $\mathbb{B} = \{B_1, B_2, \dots, B_g\}$. Each *bundle* $B \in \mathbb{B}$ is defined recursively as follows. Essentially B is a list of sorted indices of a subset of $\{1, 2, \dots, i-1\}$, but the indices are grouped in a special and systematic way.

There are two types of bundles: *atomic* and *composite*. If B has only one index, then it is an atomic bundle. Otherwise, B is a composite bundle consisting of a sequence of at least two bundles $B'_1, \dots, B'_{g'}$ (with $g' \geq 2$) such that the last bundle $B'_{g'}$ must be atomic (others can be either atomic or composite), and we call the index contained in $B'_{g'}$ the *wrap index* of B . We consider the bundles $B'_1, \dots, B'_{g'}$ as the *children bundles* of B .

Let $f_{\min}(B)$ and $f_{\max}(B)$ denote the smallest and largest $f(j)$ of all indices j of B , respectively. If B is composite, then B further has the following three *bundle-properties*. (1) The indices of B are distinct and sorted increasingly by their order in B . (2) For any $1 \leq b < g' - 1$, $f_{\max}(B'_b) < f_{\min}(B'_{b+1})$. (3) If j is the wrap index of B , then $f_{\min}(B) = f(j)$ and $\pi_{f(j)}$ crosses $w_{f(j')}$ for every $j' \in B \setminus \{j\}$ (intuitively, $\pi_{f(j)}$ “wraps” the point $q_{f(j')}$, and this is why we call j a “wrap” index). Refer to Fig. 9 for an example.

For convenience, if the context is clear, we also consider a bundle B as a set of sorted indices. So if an index j is in B , we can write “ $j \in B$ ”.

Remark. We use the word “bundle” because each index j of B refers to the shortest path $\pi_{f(j)}$. Therefore, B is a “bundle” of shortest paths.

In addition, the bundle sequence $\mathbb{B} = \{B_1, B_2, \dots, B_g\}$ maintained by our algorithm has the following two \mathbb{B} -*properties*. (1) The indices in all bundles are distinct in $[1, i-1]$ and are sorted increasingly by their order in the sequence. (2) For any $1 \leq b < g$, $f_{\max}(B_b) < f_{\min}(B_{b+1})$.

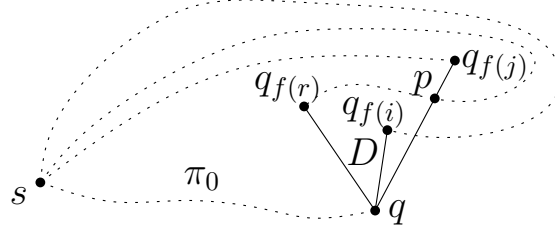


Fig. 10. Illustrating the proof of Lemma 12.

Observation 8 1. For any $1 \leq b < b' \leq g$ and any indices $j \in B_b$ and $j' \in B_{b'}$ (both B_b and $B_{b'}$ are from \mathbb{B}), the two shortest paths $\pi_{f(j)}$ and $\pi_{f(j')}$ are consistent (e.g., see Fig. 9).
 2. For any composite bundle $B = \{B'_1, \dots, B'_{g'}\}$, for any $1 \leq b < b' \leq g' - 1$ and any indices $j \in B'_b$ and $j' \in B'_{b'}$, the two shortest paths $\pi_{f(j)}$ and $\pi_{f(j')}$ are consistent (e.g., see Fig. 9).

Proof. We only prove the first part since the second part is similar.

Since $b < b'$, it holds that $j < j'$. Clearly, $f(j) \leq f_{\max}(B_b)$ and $f_{\min}(B_{b'}) \leq f(j')$. Since $b < b'$, we have $f_{\max}(B_b) < f_{\min}(B_{b'})$. Therefore, we obtain $f(j) < f(j')$. Thus, $\pi_{f(j)}$ and $\pi_{f(j')}$ are consistent. \square

In the following, we discuss our algorithm for processing the shortest path $\pi_{f(i)}$, during which \mathbb{B} will be updated. Initially when $i = 1$, we simply set \mathbb{B} to contain the only atomic bundle $B = \{1\}$ and this finishes our processing for $\pi_{f(1)}$. In general when $i > 1$, we do the following.

We first find the index β such that $f_{\max}(B_\beta) < f(i) < f_{\max}(B_{\beta+1})$. Later in Section 4.4 we will give a data structure to maintain the bundle sequence \mathbb{B} such that β can be found in $O(\log n)$ time.

If $\beta = g$ (so $B_{\beta+1}$ does not exist in this case), then we add a new atomic bundle $B_{g+1} = \{i\}$ to the rear of \mathbb{B} and we are done with processing $\pi_{f(i)}$. Note that the two \mathbb{B} -properties are maintained.

Otherwise, we check whether $f_{\min}(B_{\beta+1}) < f(i)$. We have the following lemma.

Lemma 12. *If $f_{\min}(B_{\beta+1}) < f(i)$, then the extended-window $w_{f(i)}$ can be pruned.*

Proof. Assume that $f_{\min}(B_{\beta+1}) < f(i)$. Since $f(i) < f_{\max}(B_{\beta+1})$, we have $f_{\min}(B_{\beta+1}) < f(i) < f_{\max}(B_{\beta+1})$, which also implies that $B_{\beta+1}$ is a composite bundle. Let r be the wrap index of $B_{\beta+1}$. Due to $f(r) = f_{\min}(B)$, it follows that $f(r) < f(i)$. Since every index of \mathbb{B} is smaller than i , $r < i$. By Lemma 11, $\pi_{f(r)}$ does not cross $w_{f(i)}$.

Consider the index $j \in B$ with $f(j) = f_{\max}(B)$. Hence, $f(j) > f(i)$. By the third bundle-property, $\pi_{f(r)}$ crosses $w_{f(j)}$, say, at a point p (e.g., see Fig. 10). Consider the region D bounded by $w_{f(r)}$, \overline{pq} , and the subpath of $\pi_{f(r)}$ between p and $q_{f(r)}$. Since $r < i$ and $f(r) < f(i) < f(j)$, $q_{f(i)}$ must be in D since otherwise $\pi_{f(r)}$ would cross $w_{f(i)}$, contradicting with Lemma 11(1). Also, by Observation 6, $q_{f(i)}$ is not on $\pi_{f(r)}$. Therefore, $q_{f(i)}$ is in the interior of D . This implies that the shortest path from s to any point t of $w_{f(i)}$ must intersect $w_{f(r)}$, $w_{f(j)}$, or their endpoints. Therefore, no point of $w_{f(i)}$ can be q^* . Thus, $w_{f(i)}$ can be pruned. \square

By Lemma 12, if $f_{\min}(B_{\beta+1}) < f(i)$, we simply ignore $\pi_{f(i)}$ and finish the processing of $\pi_{f(i)}$.

In the following, we assume $f(i) < f_{\min}(B_{\beta+1})$ (note that $f(i) = f_{\min}(B_{\beta+1})$ is not possible since $i \notin \mathbb{B}$). Next, we are going to find all such indices j of \mathbb{B} that $\pi_{f(j)}$ crosses $w_{f(i)}$. To this end, the following two lemmas are crucial.

Lemma 13. 1. For any index j in B_b for any $b \in [1, \beta]$, $\pi_{f(j)}$ does not cross $w_{f(i)}$.

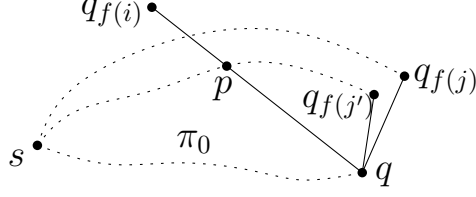


Fig. 11. Illustrating the proof of Lemma 13.

2. For any index j in B_b for any $b \in [\beta + 1, g]$, if $\pi_{f(j)}$ crosses $w_{f(i)}$, then $w_{f(j)}$ can be pruned; otherwise, $\pi_{f(i)}$ must cross $w_{f(j)}$.
3. If j is in B_b for some $b \in [\beta + 2, g]$ and $\pi_{f(j)}$ crosses $w_{f(i)}$, then $\pi_{f(j')}$ crosses $w_{f(i)}$ for any $j' \in B_{b'}$ and any $b' \in [\beta + 1, b - 1]$.
4. If j is in B_b for some $b \in [\beta + 1, g - 1]$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, then $\pi_{f(j')}$ does not cross $w_{f(i)}$ for any $j' \in B_{b'}$ and any $b' \in [b + 1, g]$.

Proof. We prove the four parts of the lemma separately.

1. If j is in a bundle B of $\{B_1, B_2, \dots, B_\beta\}$. Note that $j < i$. Since $f(j) \leq f_{\max}(B)$ and $f_{\max}(B) \leq f_{\max}(B_\beta) < f(i)$, we obtain $f(j) < f(i)$. Consequently, by Lemma 11(1), $\pi_{f(j)}$ does not cross $w_{f(i)}$.
2. If j is in a bundle B of $\{B_{\beta+1}, B_{\beta+2}, \dots, B_g\}$, then $f(j) > f(i)$. Since $j < i$, according to Lemma 11(2), either $\pi_{f(j)}$ crosses $w_{f(i)}$ or $\pi_{f(i)}$ crosses $w_{f(j)}$. If $\pi_{f(j)}$ crosses $w_{f(i)}$, by Lemma 11(2), $w_{f(j)}$ can be pruned. Otherwise, $\pi_{f(i)}$ must cross $w_{f(j)}$.
3. Let j and j' be the indices as in the lemma statement. Our goal is to show that $\pi_{f(j')}$ crosses $w_{f(i)}$. Clearly, $j' < j$ and $f(j') < f(j)$. By Lemma 11(1), $D_{f(j')}$ is contained in $D_{f(j)}$ (e.g., see Fig. 11). Since $f(i) < f(j')$ and $f(i) < f(j)$, if we move from q to $q_{f(i)}$ along $w_{f(i)}$, we will enter the interior of both $D_{f(j)}$ and $D_{f(j')}$. If we keep moving, note that we cannot encounter any point in either $w_{f(j')}$ or $w_{f(j)}$. Since $\pi_{f(j)}$ crosses $w_{f(i)}$, if we move as above on $w_{f(i)}$, we will encounter a point on $\pi_{f(j)}$, which is part of the boundary of $D_{f(j)}$. Since $D_{f(j')}$ is contained in $D_{f(j)}$, the above moving will also encounter a point p on $D_{f(j')}$ (e.g., see Fig. 11). Due to Observation 6, p cannot be $q_{f(i)}$. Hence, $\pi_{f(j')}$ must cross $w_{f(i)}$ at p .
4. This part is equivalent to the above third part.

□

For any bundle B in $\{B_{\beta+1}, B_{\beta+2}, \dots, B_g\}$, if B has two indices j and j' such that $w_{f(i)}$ crosses $\pi_{f(j)}$ but does not cross $\pi_{f(j')}$, then we say that B is a *mixed* bundle, which is necessarily a composite bundle.

Lemma 14. For any mixed bundle $B = \{B'_1, B'_2, \dots, B'_{g'}\}$, the following holds.

1. The path $\pi_{f(r)}$ must cross $w_{f(i)}$, where r is the wrap index of B , i.e., $B'_{g'} = \{r\}$.
2. If an index j is in B'_b for some $b \in [2, g' - 1]$ and $\pi_{f(j)}$ crosses $w_{f(i)}$, then $\pi_{f(j')}$ crosses $w_{f(i)}$ for any $j' \in B'_{b'}$ and any $b' \in [1, b - 1]$.
3. If an index j is in B'_b for some $b \in [1, g' - 2]$ and $\pi_{f(j)}$ does not cross $w_{f(i)}$, then $\pi_{f(j')}$ does not cross $w_{f(i)}$ for any $j' \in B'_{b'}$ and any $b' \in [b + 1, g' - 1]$.
4. If a bundle B' of B has two indices j and j' such that $w_{f(i)}$ crosses $\pi_{f(j)}$ but does not cross $\pi_{f(j')}$, then we also say that B' is a mixed bundle. This lemma applies to B' recursively.

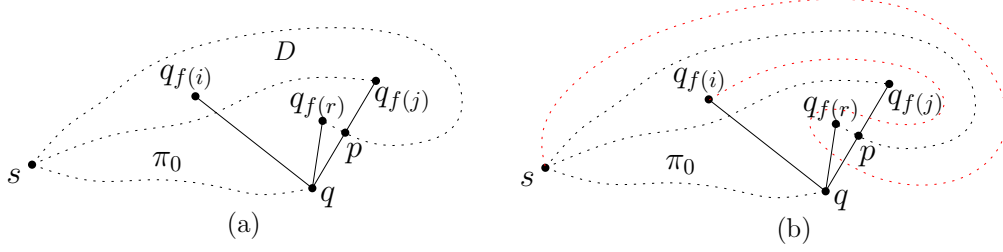


Fig. 12. Illustrating the proof of Lemma 14(1): the path $\pi_{f(i)}$ is marked with red color in (b).

Proof. 1. Suppose j is an index of B such that $\pi_{f(j)}$ crosses $w_{f(i)}$. If $j = r$, then we are done with the proof. In the following, we assume $j \neq r$. Hence, $f(j) > f(r)$.

Assume to the contrary that $\pi_{f(r)}$ does not cross $w_{f(i)}$. Since r is the wrap index, $\pi_{f(r)}$ crosses $w_{f(j)}$, say, at a point p (e.g., see Fig. 12(a)). Consider the region D bounded by $\pi_{f(j)}$, $\overline{pq_{f(j)}}$, and the subpath of $\pi_{f(r)}$ between s and p , such that D is on the right side of the directed segment $\overline{pq_{f(j)}}$ from p to $q_{f(j)}$. Since $f(i) < f(r) < f(j)$ and $w_{f(i)}$ crosses $\pi_{f(j)}$ but does not cross $\pi_{f(r)}$, $q_{f(i)}$ must be in the region D . Since $i > j$ and $i > r$, if we go from $q_{f(i)}$ to s along $\pi_{f(i)}$, we will get out of D by crossing $\overline{pq_{f(j)}}$, after which we get into the interior of the region $D_{f(j)}$ since $\pi_{f(i)}$ cannot cross $\pi_{f(r)}$ (e.g., see Fig. 12(b)). If we keep moving towards s along $\pi_{f(i)}$, before reaching s we will need to get out of the interior of $D_{f(j)}$ through $w_{f(j)}$ again. However, due to Observation 1(2), since $\pi_{f(i)}$ already crosses $w_{f(j)}$ somewhere on $\overline{pq_{f(j)}}$, it cannot intersect $w_{f(j)}$ again. Thus, we obtain contradiction.

2. This part follows the similar proof as the third part of Lemma 10 and we omit the details.
3. This part is equivalent to the second part of the lemma.
4. Using the same analysis, we can prove that the same lemma applies to B' recursively.

□

In light of the preceding two lemmas, in the following we will find the indices j of \mathbb{B} such that $\pi_{f(j)}$ crosses $w_{f(i)}$ and then prune $w_{f(j)}$ by Lemma 13(2) (i.e., remove j from \mathbb{B}); we say that such an index j is *prunable*.

Before describing our algorithm, we first discuss an operation that will be used in the algorithm. Consider a composite bundle $B = \{B'_1, B'_2, \dots, B'_{g'}\}$ of \mathbb{B} . Let r be a wrap index of B , i.e., $B'_{g'} = \{r\}$. Suppose $w_{f(i)}$ crosses $\pi_{f(r)}$. Our algorithm will remove r from B and thus from \mathbb{B} . This is done by a *wrap-index-removal* operation. Further, suppose B is the j -th bundle of \mathbb{B} , i.e., $B = B_j$. After r is removed, the operation will implicitly insert the bundles $B'_1, B'_2, \dots, B'_{g'-1}$ into the position of B in the bundle list \mathbb{B} , i.e., after the operation, \mathbb{B} becomes $B_1, \dots, B_{j-1}, B'_1, \dots, B'_{g'-1}, B_{j+1}, \dots, B_g$. Note that this new bundle list still has the two \mathbb{B} -properties. Indeed, $f_{\max}(B_{j-1}) < f_{\min}(B) = f(r) < f_{\min}(B'_1)$ and $f_{\max}(B'_{g'-1}) \leq f_{\max}(B) < f_{\min}(B_{j+1})$. Later in Section 4.4 we will give a data structure to maintain the bundles of \mathbb{B} so that each wrap-index-removal operation can be implemented in $O(\log n)$ time.

Another operation that is often used in the algorithm is the following. Given any $i, j \in [1, k]$, we want to determine whether $w_{f(i)}$ crosses $\pi_{f(j)}$. We call it the *shortest path segment intersection* (or *SP-segment-intersection*) query. Later in Section 4.6 we will present an algorithm that can answer each such query in $O(\log h \log n)$ time, after $O(n \log h)$ time and space preprocessing.

We are ready to describe our algorithm for removing all prunable indices from \mathbb{B} . By Lemma 13(1), each bundle B_b of \mathbb{B} for $1 \leq b \leq \beta$ does not contain any prunable index. For each bundle B of $B_{\beta+1}, B_{\beta+2}, \dots, B_g$ in order, we call a procedure *prune*(B) until the procedure returns “false”.

If all indices of B are prunable, then $\text{prune}(B)$ will return “true” and the entire bundle B will be removed from \mathbb{B} . Otherwise, the procedure will return false. Further, if B is a mixed bundle, then all prunable indices of B will be removed (and the procedure returns false).

The procedure $\text{prune}(B)$ works as follows (see Algorithm 1 for the pseudocode). It is a recursive procedure, which is not surprising since the bundles are defined recursively. As a base case, if B is an atomic bundle $\{j\}$, then we call an SP-segment-intersection query to check whether $\pi_{f(j)}$ crosses $w_{f(i)}$. If yes, we remove the bundle B and return true; otherwise, we return false. If B is a composite bundle $\{B'_1, B'_2, \dots, B'_{g'}\}$ with r as the wrap index (i.e., $B'_{g'} = \{r\}$), then we first call an SP-segment-intersection query to check whether $\pi_{f(r)}$ crosses $w_{f(i)}$. If not, by Lemma 14(1), B does not have any prunable index and thus we simply return false. If yes, then we call a wrap-index-removal operation to remove $B'_{g'}$. Afterwards, for each $b' = 1, 2, \dots, g' - 1$ in order, we call $\text{prune}(B'_{b'})$ recursively. If $\text{prune}(B'_{b'})$ returns false, then we return false (without calling $\text{prune}(B'_{b'+1})$). If it returns true, we remove $B'_{b'}$ (in fact all children bundles of $B'_{b'}$ have been removed by $\text{prune}(B'_{b'})$). If $b' = g' - 1$, then we return true (since all children bundles of B have been removed); otherwise, we proceed on calling $\text{prune}(B'_{b'+1})$.

Algorithm 1: The procedure $\text{prune}(B)$

Input: A bundle B

Output: remove all prunable indices of B

```

1 if  $B$  is an atomic bundle  $\{j\}$  then
2   if  $\pi_{f(j)}$  crosses  $w_{f(i)}$  then                                /* call an SP-segment-intersection query */
3     remove  $B$ ;
4     return true;
5   else
6     return false;
7 else
8   Let  $B = \{B'_1, B'_2, \dots, B'_{g'}\}$  and  $B'_{g'} = \{r\}$ ;
9   if  $\pi_{f(r)}$  does not cross  $w_{f(i)}$  then                          /* call an SP-segment-intersection query */
10    return false;
11  else
12    remove  $B'_{g'}$ ;                                                  /* perform a wrap-index-removal operation */
13    for  $b' \leftarrow 1$  to  $g' - 1$  do
14      if  $\text{prune}(B'_{b'}) = \text{false}$  then
15        return false;
16      else
17        remove  $B'_{b'}$ ;
18    return true;

```

If $\text{prune}(B_b)$ returns true for every b with $\beta + 1 \leq b \leq g$, then we add a new atomic bundle $\{i\}$ at the end of \mathbb{B} , which now becomes $\{B_1, B_2, \dots, B_\beta, \{i\}\}$. This also finishes our preprocessing for $\pi_{f(i)}$. Otherwise, $\text{prune}(B_b)$ returns false for some b with $\beta + 1 \leq b \leq g$. In this case, as a final step, we create a new composite bundle B , consisting of all bundles of \mathbb{B} after B_β (not including B_β) and the atomic bundle $\{i\}$ as the last child bundle of B . This is done by a *bundle-creation* operation. We will show in Section 4.4 that this operation can be implemented in $O(\log n)$ time. Afterwards,

the new bundle sequence \mathbb{B} becomes $\{B_1, B_2, \dots, B_\beta, B\}$. The following lemma shows that the new bundle B is a “valid” composite bundle and the updated \mathbb{B} maintains the two \mathbb{B} -properties.

Lemma 15. *The new bundle B has the three bundle properties and the updated \mathbb{B} has the two \mathbb{B} -properties.*

Proof. Let $B = \{B'_1, B'_2, \dots, B'_{g'}\}$, where $B'_{g'} = \{i\}$. We show that B has the three properties of composite bundles as follows.

1. Indeed, recall that every index of the original \mathbb{B} is smaller than i . Note that although some indices have been removed from \mathbb{B} , we never change any relative order of two indices of \mathbb{B} . Further, i is the last index of B . Therefore, the indices of B are sorted increasingly by their order in B . Hence, B has the first property.
2. To show the second property, again the bundles $B'_1, B'_2, \dots, B'_{g'-1}$, which are from the original \mathbb{B} , never change their relative orders. By the recursive definition of bundles, it holds that $f_{\max}(B'_{b'}) < f_{\min}(B'_{b'+1})$ for any $1 \leq b' < g' - 1$. Thus, the second property also holds on B .
3. For the third property, recall that $f(i) < f_{\min}(B_{\beta+1})$. Since each $B'_{b'}$ with $1 \leq b' \leq g' - 1$ is a “descendent” bundle of $B_b \in \mathbb{B}$ (we consider B_b a descendent bundle of itself) for some $b \in [\beta + 1, g]$, it holds that $f_{\min}(B_{\beta+1}) \leq f_{\min}(B_b)$. Since $f(i) < f_{\min}(B_{\beta+1})$, $f(i) < f_{\min}(B_b)$. Therefore, $f_{\min}(B) = f(i)$. Further, for each $j \in B \setminus \{i\}$, since j is not prunable (otherwise j would have already been pruned), $\pi_{f(j)}$ does not cross $w_{f(i)}$ (by Lemma 13(2)). By Lemma 13(2), $\pi_{f(i)}$ must cross $w_{f(j)}$. Hence, the third property holds on B .

To see that the updated bundle sequence \mathbb{B} maintains the two \mathbb{B} -properties, by using the similar analysis as above, the first property holds. For the second property, we have proved above that $f_{\min}(B) = f(i)$. Further, recall that $f_{\max}(B_\beta) < f(i)$. Therefore, we obtain $f_{\max}(B_\beta) < f_{\min}(B)$. Consequently, the second property also holds on \mathbb{B} . \square

To analyze the running time of the above algorithm, let m be the number of indices that have been removed from \mathbb{B} . Then, the algorithm makes at most $m + 1$ SP-segment-intersection queries. To see this, once the query discovers an index j that is not prunable, the algorithm will stop without making any more such queries. On the other hand, each wrap-index-removal operation removes an index, and thus the number of such operations is at most m . Further, observe that for each bundle B , whenever we make a recursive call on a child bundle of B , the wrap index of B is guaranteed to be removed. Therefore, the number of total recursive calls is at most m as well. Hence, the running time of the algorithm is $O((m + 1) \log h \log n)$.

This finishes our algorithm for processing the path $\pi_{f(i)}$. The total time for processing $\pi_{f(i)}$ is $O((m + 1) \log h \log n)$. Since once an index is removed from \mathbb{B} , it will never be inserted into \mathbb{B} again, the sum of all such m in the entire algorithm for processing all paths $\pi_{f(i)}$ for $i = 1, 2, \dots, k$ is at most k . Hence, the total time of the entire algorithm is $O(k \log h \log n)$.

Again, Fig. 8 gives an example showing the remaining parts of the segments of W after the pruning algorithm.

4.4 The Data Structure for Maintaining the Bundles

In this section, we give a data structure for maintaining the bundle sequence \mathbb{B} such that our algorithm runs in the time as claimed above. In particular, we show that during our algorithm for

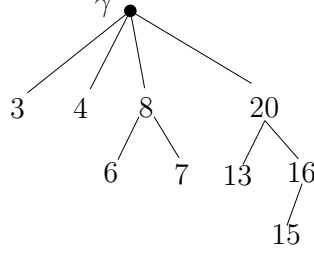


Fig. 13. Illustrating the bundle tree $T_{\mathbb{B}}$ for the bundle sequence in Fig. 9.

processing $\pi_{f(i)}$ each of the following operations can be performed in $O(\log k)$ ($= O(\log n)$) time: inserting a new bundle $\{i\}$ at the end of \mathbb{B} , the bundle-creation operation, the wrap-index-removal operation, and finding the index β .

We first present our data structure and then discuss the operations.

4.4.1 The Data Structure

Let $\mathbb{B} = \{B_1, B_2, \dots, B_g\}$. It is not difficult to see that the bundles of \mathbb{B} naturally form a tree structure. So we use a *bundle tree* $T_{\mathbb{B}}$ to represent it, as follows. The tree $T_{\mathbb{B}}$ has a root γ , whose children from left to right are exactly the bundles B_1, B_2, \dots, B_g in this order. For each such bundle B , if B is atomic, then B is a leaf of $T_{\mathbb{B}}$ and the index of B is stored at the leaf. Otherwise, suppose $B = \{B'_1, B'_2, \dots, B'_{g'}\}$. Then, we store the wrap index of B at the node B and B has $g' - 1$ children from left to right corresponding to $B'_1, B'_2, \dots, B'_{g'-1}$ in this order. If one of these bundles is composite, then its subtree is defined recursively. Refer to Fig. 13 for an example.

For each node μ of $T_{\mathbb{B}}$, let $T_{\mathbb{B}}(\mu)$ denote the subtree rooted at μ . It is easy to see that if μ is a leaf, then $T_{\mathbb{B}}(\mu)$ represents an atomic bundle; otherwise, $T_{\mathbb{B}}(\mu)$ represents a composite bundle. Each node of the tree except the root stores an index. Further, the post-order traversal of each subtree $T_{\mathbb{B}}(\mu)$ gives exactly the sequence of indices in the bundle represented by $T_{\mathbb{B}}(\mu)$.

We implement the bundle tree $T_{\mathbb{B}}$ as follows. In general, consider any internal node μ . We let μ have two pointers *front* and *rear* pointing to the leftmost and rightmost children of μ , respectively. In this way, from μ , we can access its leftmost and rightmost children in $O(1)$ time. All children of μ are organized by a doubly linked list: Each child of μ maintains a *left* (resp., *right*) pointer pointing to its left (resp. right) sibling, so that we can remove a node in constant time; the left (resp., right) pointer of the leftmost (resp., rightmost) child is empty. In this way, from the leftmost child of μ , we can visit all children of μ in order from left to right in linear time.

In order to compute the index β in $O(\log k)$ time, we use another balanced binary search tree T_f to maintain the ranges $[f_{\min}(B), f_{\max}(B)]$ of the bundles B of \mathbb{B} . The tree T_f has g leaves corresponding to the bundles of \mathbb{B} from left to right. For each leaf $v \in T_f$, let B_v denote the bundle of \mathbb{B} corresponding to v ; we associate with v the range $[f_{\min}(B_v), f_{\max}(B_v)]$. By the second property of \mathbb{B} , the ranges of the leaves from left to right are sorted by either the minimum values or the maximum values of the ranges. Clearly, the height of T_f is $O(\log k)$. In addition, each leaf v is associated with a *cross pointer* pointing to the node of $T_{\mathbb{B}}$ corresponding to the bundle B_v , so that once we have the access to v in T_f we can locate B_v in $T_{\mathbb{B}}$ in constant time. Finally, each internal node v of T_f maintains the minimum range value of the leftmost leaf in the subtree of T_f rooted at v , which is used for searching.

This completes our data structure for maintaining the bundles of \mathbb{B} , which consists of two trees $T_{\mathbb{B}}$ and T_f . In the following, we show how to use our data structure to implement the operations on \mathbb{B} needed in our algorithm for processing $\pi_{f(i)}$.

4.4.2 Performing Operations

First of all, finding the index β can be easily done in $O(\log k)$ time by searching the tree T_f . Further, by using the cross pointer, we can immediately access the node μ of $T_{\mathbb{B}}$ whose subtree $T_{\mathbb{B}}(\mu)$ represents B_{β} .

If $\beta = g$, then our algorithm adds $B = \{i\}$ at the end of \mathbb{B} . To implement it, we first insert B to T_f as the rightmost leaf with the range $[f(i), f(i)]$, which can be done in $O(\log k)$ time. Then, we add the atomic bundle B to the rear of \mathbb{B} by adding a leaf to $T_{\mathbb{B}}$ as the rightmost child of the root γ . The tree $T_{\mathbb{B}}$ can be updated in constant time with the help of the rear pointer of γ .

If $\beta \neq g$, then we check whether $f_{\min}(B_{\beta+1}) < f(i)$ (note that we can find the leaf for $B_{\beta+1}$ in T_f in $O(\log k)$ time). If $f_{\min}(B_{\beta+1}) < f(i)$, then we are done for processing $\pi_{f(i)}$. In the following, we assume $f_{\min}(B_{\beta+1}) > f(i)$.

Our algorithm first calls the procedure $\text{prune}(B_{\beta+1})$. To implement it, note that $B_{\beta+1}$ is represented by the subtree $T_{\mathbb{B}}(\mu')$, where μ' is the right sibling of μ . Since we already have the access to μ , by using the right pointer of μ , we can access μ' in constant time. The procedure $\text{prune}(B_{\beta+1})$ begins with checking whether $B_{\beta+1}$ is atomic, which can be done in constant time by checking whether μ' is a leaf.

If yes, then the procedure stops after an SP-segment-intersection query. Further, if $B_{\beta+1}$ needs to be removed, then we simply remove the leaf μ' , which can be done in constant time (recall that the children of any node of $T_{\mathbb{B}}$ are organized by a doubly linked list). Further, we also remove the corresponding leaf from T_f in $O(\log k)$ time.

If $B_{\beta+1}$ is not atomic, let $B_{\beta+1} = \{B'_1, B'_2, \dots, B'_{g'}\}$. We can obtain the wrap index of $B_{\beta+1}$ in constant time since it stored at the node μ' . To implement wrap-index-removal operation, essentially, we need to replace the node μ' by its children. This can be done in constant time by using the left, right, front, and rear pointers of μ' . Depending on whether μ' is the leftmost or rightmost child of γ , we may also need to update the front or rear pointer of γ , which can also be easily done in constant time. We omit these details.

Next, our algorithm calls the procedure $\text{prune}(B'_1)$. We can access the node of $T_{\mathbb{B}}$ whose subtree represents B'_1 in constant time after the above wrap-index-removal operation (i.e., by following the front pointer of μ'). The algorithm then works recursively. Note that B'_1 now becomes a bundle of \mathbb{B} . Hence, the above algorithm description on $B_{\beta+1}$ applies to B'_1 recursively.

The algorithm stops when either we are at the end of \mathbb{B} or the procedure $\text{prune}(B')$ returns false for a bundle B' in the current \mathbb{B} . In the former case, we add $\{i\}$ to the rear of the current list \mathbb{B} in the same way as before. In the latter case, we perform a bundle creation operation by creating a composite bundle B including all bundles of the current \mathbb{B} after B_{β} as well as $\{i\}$ in the rear of B . We implement this bundle creation operation as follows.

Note that we have the access of the node μ_1 whose subtree represents B' after $\text{prune}(B')$ returns false. Let μ_2 be the rightmost child of γ , which can be accessed in constant time from the root γ . Next, in constant time, we construct a subtree T representing the bundle B and use T to replace the subtrees of γ from μ_1 to μ_2 (e.g., see Fig. 14), as follows. First, we create a new node μ_3 storing the single index i . Second, we set the front pointer of μ_3 to μ_1 and set the rear pointer of μ_3 to μ_2 . Third, if μ_1 has a left sibling, denoted by μ_4 , then we set the left pointer of μ_3 to μ_4 and set

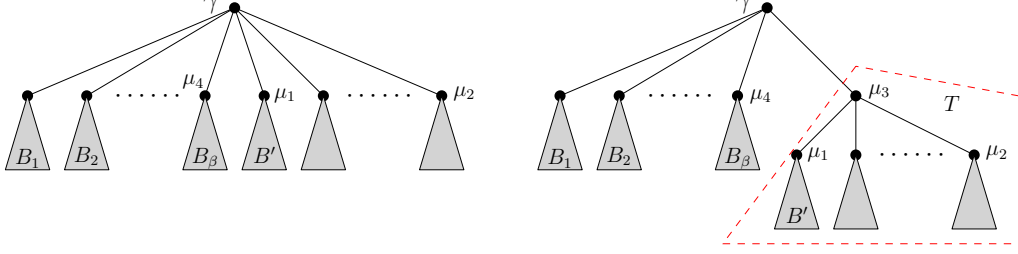


Fig. 14. Illustrating the bundle creation operation. Left: the bundle tree before the operation. Right: the bundle tree after the operation (the subtree T represents the bundle B).

the right pointer of μ_4 to μ_3 ; otherwise, we set the front pointer of γ to μ_3 . Fourth, we set the rear pointer of γ to μ_3 . Fifth, we set the left pointer of μ_1 to empty.

Finally we update the tree T_f as follows. Recall that the algorithm stops when either we are at the end of \mathbb{B} or $\text{prune}(B')$ returns false for a bundle B' in the current \mathbb{B} . In the former case, we let $B = \{i\}$, and in the latter case, we let B denote the new bundle created by the bundle creation procedure. In either case, we update T_f as follows. Note that the original \mathbb{B} is $\{B_1, B_2, \dots, B_g\}$ and the updated \mathbb{B} is $\{B_1, B_2, \dots, B_\beta, B\}$. Essentially, the bundles $\{B_{\beta+1}, B_{\beta+2}, \dots, B_g\}$ have been replaced by B . So we first remove the leaves corresponding to the bundles $\{B_{\beta+1}, B_{\beta+2}, \dots, B_g\}$ from T_f . Since they are consecutive in T_f , the remove can be done in $O(\log k)$ time. Next, we insert B into T_f as the rightmost leaf. In the former case (i.e., $B = \{i\}$), $f_{\min}(B) = f_{\max}(B) = f(i)$. In the latter case, $f_{\min}(B) = f(i)$ and $f_{\max}(B) = f_{\max}(B_g)$, which can be obtained in $O(\log k)$ time from the original T_f . Hence, in either case the total time for updating T_f is $O(\log k)$. In addition, we set the cross pointer of the new leaf to the node μ'' of $T_{\mathbb{B}}$ whose subtree represents B , which is done in constant time since we have the access of μ'' after $T_{\mathbb{B}}$ is updated (e.g., μ'' is μ_3 in the case of Fig. 14).

4.5 Computing the Closest Point q^*

Recall that we have assumed that q^* is on w_i^l for some $i \in [1, k]$, i.e., $q^* = q_i^*$. According to our pruning algorithm for computing the bundle sequence \mathbb{B} , the point q^* must be on $w_{f(j)}^l$ for some index $j \in \mathbb{B}$. In this section, we will compute q^* by using the bundle sequence \mathbb{B} . For example, in Fig 8, our goal is to compute q^* on the left sides of those (red) thick segments.

Recall that we have defined in Section 3.2 that R_i is the region of \mathcal{P} bounded by $\pi(s, v_i)$, $\pi(s, v_{i+1})$, and α_i , where α_i is either a bisector super-curve whose endpoints are v_i and v_{i+1} or a chain of obstacle edges. Also recall that R_i consists of a tail and a cell.

Let τ be any segment in \mathcal{P} such that R_i contains $\pi(s, \tau)$. With the help of the decomposition \mathcal{D} proposed in Section 3, we propose a *region-processing* algorithm to compute $\pi(s, \tau)$ in the following lemma.

Lemma 16. *Suppose τ is a segment of \mathcal{P} such that R_i contains $\pi(s, \tau)$ and R_i is known. Then $\pi(s, \tau)$ can be computed in $O(\log h \log n)$ time, after $O(n \log h)$ time and space preprocessing.*

Proof. We first present our region-processing algorithm for computing $\pi(s, \tau)$, and then argue its correctness. Finally, we will analyze the running time of the algorithm.

The algorithm. For each of $\pi(s, v_i)$, $\pi(s, v_{i+1})$, and α_i , we check whether it crosses τ . Note that this step is not necessary for α_i if α_i is a chain of obstacle edges since τ cannot cross any obstacle edge. By Observation 1(2), τ intersect $\pi(s, v_i)$ (resp., $\pi(s, v_{i+1})$) at most once.

To avoid the tedious case analysis, by Observation 1(2), we assume that if τ intersects $\pi(s, v_i)$ or $\pi(s, v_{i+1})$, then the intersection is a single point (i.e., not a general sub-segment of τ). Let a (resp., b) be the intersection between τ and $\pi(s, v_i)$ (resp., $\pi(s, v_{i+1})$); if there is no intersection, we simply let a (resp., b) refer to \emptyset . In general, if α_i is a bisector super-curve, τ may intersect α_i multiple times, and we let c be an arbitrary such intersection; similarly, if there is no intersection let c refer to \emptyset .

If $a = b$ and $a \neq \emptyset$, then a is a point on the tail of R_i . By Observation 1(2), τ can only intersect the tail once. By the definition of R_i , for any point t in the cell of R_i , $d(s, a) \leq d(s, t)$. This implies that $\pi(s, a)$ is $\pi(s, \tau)$. So we can finish the algorithm in this case.

Otherwise (i.e., $a \neq b$ or $a = b = \emptyset$), if at least one element of $\{a, b, c\}$ is not \emptyset , then for each point p of $\{a, b, c\}$ and $p \neq \emptyset$, we do the following. Observe that p is not on the tail of R_i . By the definition of the decomposition \mathcal{D} , regardless of whether p is on $\pi(s, v_i)$, $\pi(s, v_{i+1})$, or α_i , there is a cell Δ_p of \mathcal{D} such that Δ_p contains p and Δ_p is in R_i . By Lemma 1(4), $\Delta_p \cap \tau$ consists of at most two maximal sub-segments τ_1 and τ_2 . Since Δ_p is a simple polygon, we can build a ray-shooting data structure on each of the inside and the outside of Δ_p . Then, we can compute τ_1 and τ_2 in $O(\log n)$ time by using ray-shooting queries. Next, we compute $\pi(s, \tau_1)$ and $\pi(s, \tau_2)$ in $O(\log n)$ time by Lemma 1(5). In this way, we obtain at most six candidate paths (for the at most three non-empty points of $\{a, b, c\}$) and return the shortest one as $\pi(s, \tau)$.

The remaining case is when every element of $\{a, b, c\}$ is \emptyset , i.e., τ does not cross any of the three parts of ∂R_i . In this case, τ is contained in a single cell Δ of \mathcal{D} . We can determine Δ by locating the cell of \mathcal{D} that contains an arbitrary endpoint of τ . Then, we compute $\pi(s, \tau)$ by Lemma 1(5).

The correctness. Recall that R_i contains $\pi(s, \tau)$. Let t a closest point of τ (i.e., $\pi(s, \tau) = \pi(s, t)$). Thus, R_i contains t . If t is on the tail of R_i , then our algorithm correctly computes $\pi(s, \tau)$ as discussed above. Otherwise, if τ is in R_i , then τ must be in a single cell of \mathcal{D} . Clearly, our algorithm correctly computes $\pi(s, \tau)$ in this case. If τ is not in R_i , then since R_i contains t , τ must cross the boundary of R_i . Suppose we move from t along τ until we cross the boundary of R_i at a point p . Let Δ_p be the cell of \mathcal{D} that is in R_i and contains p . By definition, Δ_p also contains t . If p is on $\pi(s, v_i)$ (resp., $\pi(s, v_{i+1})$), then since τ intersects $\pi(s, v_i)$ (resp., $\pi(s, v_{i+1})$) at a single point, our algorithm correctly computes $\pi(s, \tau)$. If p is on α_i , then all intersections between τ and α_i are in Δ_p since α_i is contained in Δ_p . Hence, our algorithm also correctly computes $\pi(s, \tau)$.

The time analysis. The algorithm needs at most six calls of Lemma 1(5), which take $O(\log n)$ time. It also has at most two SP-segment-intersection queries for computing the intersections of τ with $\pi(s, v_i)$ and $\pi(s, v_{i+1})$. Again, we will show that each such query can be answered in $O(\log h \log n)$ time with $O(n \log h)$ time and space preprocessing.

In addition, if α_i is a bisector super-curve, our algorithm also needs to compute an intersection between τ and α_i . This can be done in $O(\log n)$ time after linear time preprocessing on α_i using the ray-shooting data structure on curved simple polygons or splinegons [26] (indeed, each bisector edge of α_i is convex, and thus it is straightforward to make α_i a splinegon [26], e.g., by the standard technique as detailed in the proof of Lemma 20). Thus, the total preprocessing time on all such curves α_i for $i = 1, 2, \dots, h^*$ is $O(n)$.

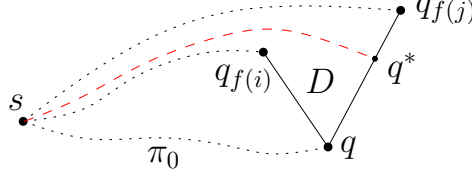


Fig. 15. Illustrating Observation 9.

Also, we have mentioned before that we need a constant number of ray-shooting queries on the cells Δ_p to determine the at most two sub-segments of $\Delta_p \cap \tau$. The query time is $O(\log n)$ and the total preprocessing time on all cells of \mathcal{D} is $O(n)$.

Hence, our region-processing algorithm runs in $O(\log h \log n)$ time, and the total preprocessing time and space is $O(n \log h)$. \square

Recall that $\mathcal{R} = \{R_1, R_2, \dots, R_{h^*}\}$. Due to our general position assumption that q is not collinear with any two obstacle vertices, none of $\{q, q_1, \dots, q_k\}$ is an obstacle vertex. Then, for each $k' \in [0, k]$, there is a unique region R_i of \mathcal{R} whose cell contains $q_{f(k')}$, such that the shortest path $\pi_{f(k')}$ is contained in R_i , and we let $z(k')$ refer to the index i of R_i . Computing $z(0), z(1), \dots, z(k)$ can be done in $O(k \log n)$ time by point location queries on the cells of the regions of \mathcal{R} .

For any two indices k_1 and k_2 in $[1, h^*]$, if $k_1 \leq k_2$, then let $[k_1, k_2]_R$ denote the set of all integers $k' \in [k_1, k_2]$; otherwise, let $[k_1, k_2]_R$ denote the set of all integers $k' \in [k_1, h^*] \cup [1, k_2]$. Recall that the regions R_1, R_2, \dots, R_{h^*} are counterclockwise around s . We actually use $[k_1, k_2]_R$ to refer to the set of indices of the regions of \mathcal{R} from R_{k_1} to R_{k_2} counterclockwise around s .

Next we compute q^* on $w_{f(j)}^l$ for $j \in \mathbb{B}$, by using our region-processing algorithm in Lemma 16. Consider the bundles of $\mathbb{B} = \{B_1, B_2, \dots, B_g\}$. For each b with $1 \leq b \leq g$, we call a procedure $path(B_b, z(i))$, where i is the last index of B_{b-1} if $b \geq 2$ and $i = 0$ otherwise. Note that given the access of B_b , we can obtain i in constant time by using our data structure in Section 4.4. Also note that $i < j$ for any index $j \in B_b$. The procedure $path(B_b, z(i))$ works as follows.

Depending on whether B_b is atomic or composite, there are two cases.

The atomic case. If B_b is atomic, let j be the only index of B_b . According to the bundle-properties, $i < j$ and $f(i) < f(j)$. So $\pi_{f(j)}$ and $\pi_{f(i)}$ are consistent. By Lemma 11(1), D_i is contained in D_j . Let D be D_j minus the interior of D_i . We have the following observation.

Observation 9 *If q^* is on $w_{f(j)}^l$, then $\pi(s, q^*)$ must be in D (e.g., see Fig. 15).*

Proof. Suppose q^* is on $w_{f(j)}^l$. Let $t = q^*$. By definition, the point t^+ is in the interior of D . Since $t = q^*$, $\pi(s, t^+)$ does not intersect any point of $w_{f(i)}$ or $w_{f(j)}$ and it does not contain q either. Also, $\pi(s, t^+)$ cannot cross either $\pi_{f(i)}$ or $\pi_{f(j)}$. Hence, $\pi(s, t)$ must be in D . \square

Observation 9 leads to the following lemma.

Lemma 17. *If q^* is on $w_{f(j)}^l$, then $\pi(s, q^*)$ is in $R_{k'}$ for some index $k' \in [z(i), z(j)]_R$, and further, any shortest path $\pi(s, w_{f(j)})$ from s to $w_{f(j)}$ is $\pi(s, q^*)$.*

Proof. Suppose q^* is on $w_{f(j)}^l$. Since q^* is also a closest point of $w_{f(j)}$, $\pi(s, w_{f(j)})$ must be $\pi(s, q^*)$.

Note that $\pi(s, q^*)$ must be contained in a region of \mathcal{R} . By Observation 9, $\pi(s, q^*)$ is in D . Hence, $\pi_{f(j)}$ is counterclockwise from $\pi(s, q^*)$ with respect to $\pi_{f(i)}$ around s . Since $\pi_{f(j)}$ is in $R_{z(j)}$, and

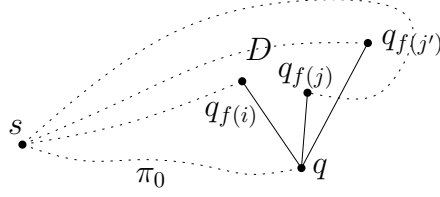


Fig. 16. j is the wrap index of B_b and j' is another index of B_b with $j' \neq j$; $\pi_{f(j')}$ is in the region D .

$\pi_{f(i)}$ is in $R_{z(i)}$, there is a region $R_{k'} \in \mathcal{R}$ that contains $\pi(s, q^*)$ such that $R_{z(j)}$ is counterclockwise from $R_{k'}$ with respect to $R_{z(i)}$ around s , which implies that $k' \in [z(i), z(j)]_R$. \square

For each $k' \in [z(i), z(j)]_R$, we apply our region-processing algorithm on $R_{k'}$ and $w_{f(j)}$ to obtain a path, and we keep the shortest path π among all such paths; let $q_{f(j)}^l$ be the endpoint of π on $w_{f(j)}$. According to Lemma 17, if q^* is on $w_{f(j)}^l$, then q^* must be $q_{f(j)}^l$.

For the purpose of analyzing the total running time of our algorithm, as will be seen later, for each $k' \in [z(i), z(j)]_R$ with $k' \neq z(i)$ and $k' \neq z(j)$, the region-processing algorithm will not be called on $R_{k'}$ again in the entire algorithm for computing q_i^* . On the other hand, we charge the two algorithm calls on $R_{k'}$ for $k' = z(i)$ and $k' = z(j)$ to the index j of \mathbb{B} . In this way, the total number of calls to the region-processing procedure in the entire algorithm is $O(h^* + k)$ since the total number of indices of \mathbb{B} is at most k and the total number of regions $R_{k'}$ is h^* .

The composite case. If B_b is composite, the algorithm is more complicated. Let j be the wrap index of B_b . Observation 9 and Lemma 17 still hold on j . However, since now the region D also contains a portion of $w_{f(j')}$ for each $j' \in B_b$ and $j' \neq j$ (e.g., see Fig. 16), D may also contain the shortest path from s to $w_{f(j')}$. In order to avoid calling the region-processing procedure on the same region of \mathcal{R} too many times, we use the following approach to process $w_{f(j)}$.

For any two different indices of k' and k'' in a range $[k_1, k_2]_R$ of indices of the regions of \mathcal{R} , we say that k'' is *ccw-larger* than k' if $[k', k'']_R$ is a subset of $[k_1, k_2]_R$ (e.g., if $k_1 < k_2$, then $k' < k''$).

Define z_{ij} to be the ccw-largest index in $[z(i), z(j)]$ such that $w_{f(j)}$ crosses $\partial R_{z_{ij}}$ (if no such index exists, then let $z_{ij} = z(i)$). We first compute z_{ij} (to be discussed later). Then, we call the region-processing procedure on $R_{k'}$ for all $k' \in [z(i), z_{ij}]$ and return the shortest path π that is found; let $q_{f(j)}^l$ be the endpoint of π on $w_{f(j)}$. By the following lemma, if q^* is on $w_{f(j)}^l$, then $q_{f(j)}^l$ is q^* .

Lemma 18. *If q^* is on $w_{f(j)}^l$, then $\pi(s, q^*)$ is in $R_{k'}$ for some index $k' \in [z(i), z_{ij}]_R$, and further, any shortest path $\pi(s, w_{f(j)})$ from s to $w_{f(j)}$ is $\pi(s, q^*)$.*

Proof. By Lemma 17, the lemma statement holds for some $k' \in [z(i), z(j)]_R$. In the following we show that k' is in $[z(i), z_{ij}]_R$.

Assume to the contrary that k' is not in $[z(i), z_{ij}]_R$. Then, k' is ccw-larger than z_{ij} and $w_{f(j)}$ does not cross $\partial R_{k'}$. This implies that $w_{f(j)}$ and q are in $R_{k'}$. Since $i < j$, $\pi_{f(j)}$ is counterclockwise from $\pi_{f(i)}$ with respect to $\pi_0 = \pi(s, q)$. This implies that $z(i) \in [z(0), z(j)]_R$. But $w_{f(j)} \in R_{k'}$ implies that $z(0) = z(j) = k'$. Thus, we have $z(i) = z(j) = k'$. Since $z_{ij} \in [z(i), z(j)]_R$, we obtain $z_{ij} = k'$. But this contradicts with that k' is not in $[z(i), z_{ij}]_R$.

The lemma thus follows. \square

After $w_{f(j)}$ is processed for each $j \in \mathbb{B}$, $q_{f(j)}^l$ is computed for every $j \in \mathbb{B}$; among these at most k points, we return the point q' whose value $d(s, q')$ is the smallest as q_l^* , which is q^* based on our above analysis (and also due to our assumption that q^* is on w_i^l for some $i \in [1, k]$). The total number of calls on the region-processing procedures is $O(k + h^*)$. The total number of \mathcal{R} -region range queries is $O(k)$ since each such query is for a composite bundle and there are at most k bundles in total. Hence, the total time of the algorithm is $O((h + k) \log h \log n)$. Recall that $k \leq K$.

4.6 The Algorithm Implementation

In this section, we discuss some implementation details left out above. Specifically, we will give our algorithm for computing the map $f(\cdot)$, and give our data structures for answering the SP-segment-intersections queries and the \mathcal{R} -region range queries.

4.6.1 Computing the Map $f(\cdot)$

Recall the definitions of Q , \mathcal{C}_Q , and \mathcal{L}_Q in Section 4.2. Computing the map $f(\cdot)$ is to compute the list $\mathcal{L}_Q = \{q, q_{f(1)}, \dots, q_{f(k)}\}$. Intuitively, we want to order the paths π_1, \dots, π_k counterclockwise around s with respect to π_0 . Our goal is to prove Lemma 8.

We begin with our preprocessing algorithm. Let $\Sigma(s)$ denote the decomposition of $SPM(s)$ by the edges of $SPT(s)$, which can be constructed in $O(n)$ time after $SPM(s)$ is given. For each cell σ of $\Sigma(s)$, we pick an arbitrary point in the interior of σ as the *representative point* of σ . Let X denote the set of all such representative points. Let T_X be the tree that is the union of the shortest paths from s to all points of X , and let s be the root of T_X . Clearly, T_X has $O(n)$ nodes and can be computed in $O(n)$ time once we have $\Sigma(s)$. The points of X are exactly the leaves of T_X . We find a base leave p^* of T_X in $O(n)$ time. Then, we compute in $O(n)$ time the list $\mathcal{L}_l(T_X, p^*)$ of all leaves and the cycle $\mathcal{L}_l(T_X)$. To simplify the notation, let $\mathcal{L}_X = \mathcal{L}_l(T_X, p^*)$ and let $\mathcal{C}_X = \mathcal{L}_l(T_X)$. This finishes our preprocessing, which takes $O(n)$ time.

In the sequel, we discuss our algorithm for computing the list \mathcal{L}_Q in $O(k \log n)$ time. It is sufficient to compute the circular list \mathcal{C}_Q since we can obtain \mathcal{L}_Q from \mathcal{C}_Q in $O(k)$ time by breaking the cycle at q .

Let $q_0 = q$ (temporarily only for the discussion in this subsection). Recall that for each point $q_i \in Q$ with $0 \leq i \leq k$, u_i is the root of the cell of $SPM(s)$ that contains q_i and determines the shortest path π_i , and note that $\overline{q_i u_i}$ is in a cell of $\Sigma(s)$, denoted by σ_i (which can be determined in $O(\log n)$ time by a point location in $\Sigma(s)$). If all cells $\sigma_0, \sigma_1, \dots, \sigma_k$ are distinct, then the order of the points of Q following the relative order of the representative points of the cells $\sigma_0, \sigma_1, \dots, \sigma_k$ in \mathcal{C}_X is exactly \mathcal{C}_Q , which can be computed in $O(k \log n)$ time with help of the circular list \mathcal{C}_X .

If $\sigma_0, \sigma_1, \dots, \sigma_k$ are not distinct, then we first compute the circular list of the cells by the above algorithm. To simplify the notation, let $\sigma_0, \sigma_1, \dots, \sigma_k$ be the circular list. Then, two cells are the same only if they are adjacent in the list. Hence, we can determine in $O(k)$ time the cycle of unique cells $\sigma'_0, \sigma'_1, \dots, \sigma'_{k'}$ for $k' < k$, and further, for each cell σ'_i , the set $Q(\sigma'_i)$ of points of Q in σ'_i can also be determined. Consider a cell σ'_i and let u'_i be the root. Let $T(\sigma'_i)$ be the union of the segments $\overline{u'_i q'}$ for all $q' \in Q(\sigma'_i)$, and we consider $T(\sigma'_i)$ as a tree rooted at u'_i . Since u'_i is an obstacle vertex, u'_i is a node in T_X . If u'_i is not s , then let p be the parent of u'_i in T_X ; otherwise let p be the child of s in T_X that is an ancestor of the base leave p^* (we compute that particular child of s in the preprocessing). Starting from the counterclockwise first child of u'_i in $T(\sigma'_i)$ with respect to $\overline{u'_i p}$, and let $\mathcal{L}(\sigma'_i)$ be the list of the children of u'_i in $T(\sigma'_i)$ ordered counterclockwise. It can be verified that

the concatenation of $\mathcal{L}(\sigma'_0), \mathcal{L}(\sigma'_1), \dots, \mathcal{L}(\sigma'_{k'})$ is exactly the circular list \mathcal{C}_Q . Following the above description, the circular list \mathcal{C}_Q can be computed in $O(k \log n)$ time.

This proves Lemma 8.

4.6.2 The SP-segment-intersection Queries

In this section, we present our data structure for answering the SP-segment-intersection queries. Specifically, given any $i, j \in [1, k]$, we want to determine whether $w_{f(i)}$ crosses $\pi_{f(j)}$, and if yes, compute an intersection. Here we consider a more general problem. Given a point t and a segment τ in \mathcal{P} , we want to compute an intersection between τ and the shortest path $\pi(s, t)$ (or report none if they do not intersect). In the case where t has multiple shortest paths (and thus $\pi(s, t)$ is not unique), the root r of a cell of $SPM(s)$ should also be provided so that $\pi(s, t)$ refers to the one that contains \overline{rt} . But to simplify the discussion, we assume t always has a unique shortest path (the other case can be solved by our algorithm too).

We will show that with $O(n \log h)$ time and space preprocessing (with a given $SPM(s)$), each such query can be answered in $O(\log h \log n)$ time. When $h = O(1)$, the result is optimal.

Recall the definitions of V , Π , T_V , and the list $\mathcal{L}_l(T_V, v_1) = \{v_1, v_2, \dots, v_{h^*}\}$ in Section 3. In the following, we build up our data structure incrementally: We will first show how to answer queries when t is in V , then show how to answer queries when t a vertex of T_V , and finally discuss the general case where t can be any point in \mathcal{P} .

We build a complete binary search tree T_1 as follows. The leaves of T_1 from left to right correspond to the points v_1, v_2, \dots, v_{h^*} of V in this order. In the following we will consider the points of V and the leaves of T_1 interchangeably. Note that each point of V is also a leaf in the tree T_V . Consider any node u of T_1 . We maintain a path $P(u)$ of edges of T_V , defined as follows. Let $T_1(u)$ be the subtree of T_1 rooted at u and let $S(u)$ be the set of the leaves of $T_1(u)$. If u is the root, then $P(u)$ is the common sub-path (i.e., the intersection) of the shortest paths $\pi(s, p)$ for all $p \in S(u)$ (note that $\pi(s, p)$ is also the path of T_V from p to the root s). Otherwise, $P(u)$ is the portion of the common sub-path of $\pi(s, p)$ for all $p \in S(u)$ that is not stored in $P(u')$ for any ancestor u' of u . In this way, for each leaf v_i , the edges of $P(u)$ of all nodes u in the path of T_1 from v_i to the root are pairwise disjoint and comprise exactly $\pi(s, v_i)$. Further, for each node u of T_1 , since $P(u)$ is a path of edges, we build a ray-shooting data structure on $P(u)$ by standard techniques as detailed in the following lemma.

Lemma 20. *For the path $P(u)$ of each node u of T_1 with $m = |P(u)|$, we can build a data structure of $O(m)$ size in $O(m)$ time such that given any ray ρ in the plane, we can compute in $O(\log m)$ time the first intersection (if any) between ρ and $P(u)$.*

Proof. This can be easily done by using the ray-shooting data structure for simple polygons [6,21]. We provide the details below.

Let R be a big rectangle in the plane that contains all edges of $P(u)$. Let p be the topmost point of $P(u)$. We shoot a ray from p upwards until it hits ∂R at a point p' . Then, we can consider $P(u)$, $\overline{pp'}$, and R bounds a simple polygon P . We build a ray-shooting data structure in P in $O(m)$ size and space [6,21].

Consider any ray-shooting query for $P(u)$. Given a ray ρ , we compute the first point a of ∂P hit by ρ in $O(\log m)$ time by using the ray-shooting data structure on P . If a is on $P(u)$, then we are done and return a as the answer. If a is on ∂R , then we are also done and report that there is

no intersection between ρ and $P(u)$. If a is on $\overline{pp'}$, then we keep shooting the ray after a and using the ray-shooting data structure again to compute the next point $a' \in \partial P$ hit by the ray. Similarly as above, if a' is on $P(u)$, then we are done and return a' . If a' is on ∂R , then we report that there is no intersection. Note that a' cannot be on $\overline{pp'}$. Hence, we can answer the ray-shooting query on $P(u)$ in $O(\log m)$ time by making at most two ray-shooting queries on P . \square

We call the information associated with each node u of T_1 the *auxiliary data structure* at u .

Lemma 21. *The size of T_1 is $O(n \log h)$ and T_1 can be built in $O(n \log h)$ time.*

Proof. Recall that the number of edges of T_V is $O(n)$. In the following, we first show that each edge e of T_V is stored in $P(u)$ of at most two nodes u in each level of T_1 .

Assume to the contrary that there are three such nodes u in the same level of T_1 that all store the same edge e of T_V in $P(u)$. Let the three nodes be u_1, u_2, u_3 from left to right. If u_1, u_2, u_3 are consecutive, then two of them, say, u_1 and u_2 , must share the same parent u' . Since e is in both $P(u_1)$ and $P(u_2)$, by definition, e should be in $P(u')$ for an ancestor u' of u (including u itself). Thus, e should not be in either $P(u_1)$ or $P(u_2)$, incurring contradiction.

In the following we assume u_1, u_2, u_3 are not consecutive. If two of them share the same parent, then we can apply the same argument as above. Otherwise, we show below that the sibling u' of u_2 (i.e., u and u' share the same parent) has $P(u')$ including e . Consequently, the above proof applies.

Let V_e be the set of points of V whose paths from s in T_V contain the edge e . Note that V_e consists of exactly the leaves in the subtree of T_V separated by e . By the definition of $\mathcal{L}_l(T_v, v_1)$, the points of V_e are consecutive in $\mathcal{L}_l(T_v, v_1) = \{v_1, v_2, \dots, v_{h^*}\}$. According to the definition of T_1 , the leaves of T_1 corresponding to the points of V_e are consecutive in T_1 . Since e is in both $P(u_1)$ and $P(u_3)$, all leaves of the subtrees of $T_1(u_1)$ and $T_1(u_3)$ are in V_e . Since u_2 is between u_1 and u_3 , u' is also between u_1 and u_2 . Thus, all leaves of $T_1(u')$ must also be in V_e , implying that e is in the common sub-path of $\pi(s, p)$ for all $p \in S(u')$. Since e is in $P(u_2)$, e is not in $P(u'')$ for any proper ancestor u'' of u_2 . Because u' and u_2 share the same parent, we obtain that e is also in $P(u')$.

This proves that each edge e of T_V is stored in at most two nodes in each level of T_1 . Since T_1 has $O(\log h^*)$ levels and $h^* = O(\log h)$, each edge e is stored in $O(\log h)$ nodes. Hence, the size of T_1 is $O(n \log h)$.

In the following, we construct the tree T_1 in $O(n \log h)$ time. The key is to compute $P(u)$ for each node u of T_1 , after which constructing the ray-shooting data structure on $P(u)$ can be done in linear time by Lemma 20.

For each edge e of T_V , we compute the range $[l_e, r_e] \subseteq [1, h^*]$ that consists of all indices i such that e is contained in the path from v_i to s in T_V . This can be done in $O(n)$ time as follows. For each vertex v of T_V , we define the range $[l_v, r_v]$ as the set of all indices i such that v is contained in the path from v_i to s in T_V . We first compute the ranges for all vertices of T_V . This can be easily done a post-order traversal of T_V starting from the leaf v_1 . Specifically, during the traversal for each vertex v , if v is a leaf containing $v_i \in V$, we set $l_v = i$ and $r_v = i$; otherwise, all children of v have been visited and we set l_v (resp., r_v) to be the smallest (resp., largest) $l_{v'}$ of all children v' of v . After the traversal, the ranges for all vertices of T_V are computed. Then, for each edge e of T_V , it is not difficult to see that the range of e is the same as that of v , where v is the endpoint of e such that the path from s to v in T_V contains e .

Next we compute $P(u)$ for all nodes u of T_1 as follows. We consider the edges of T_V following the post-order traversal from v_1 . For each edge e , by using the range $[l_e, r_e]$, we find those nodes u of T_1 whose $P(u)$ contains e . This can be done in the similar way as the standard insertion operation in

segment trees [4]. Specifically, for each node u of T_1 , let $[l_u, r_u]$ be the range consists of all indices i such that v_i is $S(u)$. Starting from the root of T_1 , for each node u , if $[l_u, r_u] \subseteq [l_e, r_e]$, then we insert e to $P(u)$; otherwise, for each child u' of u , if $[l_e, r_e] \cap [l_{u'}, r_{u'}] \neq \emptyset$, then we proceed on u' recursively. As the standard insertion operations on segment trees, each edge e is processed in $O(\log h)$ time since the height of T_1 is $O(\log h)$. Hence, the total time of the algorithm is $O(n \log h)$. Note that since we consider the edges of T_V by following the post-order traversal from v_1 , whenever we insert an edge e to $P(u)$, e is always the edge adjacent to the first edge of the current $P(u)$ and e is then appended to $P(u)$ as the new first edge. After the algorithm finishes, the sub-path $P(u)$ is readily available by following the edges in the order they have been inserted and the first edge is the one closest to s .

This proves the lemma. \square

We show how to answer SP-segment-intersection queries by using the tree T_1 . We begin with a special case where the query point t is in V , say $t = v_i$ for some $i \in [1, h^*]$. Our goal is to compute an intersection between τ and $\pi(s, v_i)$. To answer the query, we follow the path of T_1 from the root to the leaf v_i . For each node u in the path, we use a ray-shooting query to compute an intersection between $P(u)$ and τ . If we find an intersection, then we report the intersection and stop the algorithm; otherwise, we proceed on the next node. The correctness of the algorithm is based on the fact that the union of $P(u)$ of all nodes u in the above path is exactly $\pi(s, v_i)$. The query time is $O(\log h \log n)$ since each ray-shooting query takes $O(\log n)$ time and the height of T_1 is $O(\log h)$.

We then consider a more general case where the query point t is a vertex v of T_V (v is not necessarily in V). To answer the query, we first pick an arbitrary leaf v_i in the subtree of T_V rooted at v (for this, in the preprocessing step we need to associate with v' an arbitrary leaf in its subtree for each node v' of T_V). Clearly, v must be in the path $\pi(s, v_i)$. We follow the path of T_1 from the root to the leaf v_i . For each node u in the path, we compute an intersection between $P(u)$ and τ by using a ray-shooting query. If there is an intersection p , we check whether p is in the sub-path of $\pi(s, v_i)$ between s and v (see below for more details about this). If yes, then we report p and stop the algorithm. Otherwise, since τ can only cross $\pi(s, v_i)$ once, there cannot be any intersection between τ and $\pi(s, v)$; thus, in this case we simply return none. If there is no intersection between τ and $P(u)$, then we proceed on the next node in the path. If we do not find any intersection after we reach v_i , then we report none.

It remains to discuss how to determine whether p is between s and v . The point p is on an edge e of $\pi(s, v_i)$, which is also in T_V . Let v' be the endpoint of e that is farther to s in T_V . Observe that p is between s and v if and only if v' is between s and v . To determine the latter, observe that v' is between s and v if and only if v' is after v in the canonical list $\mathcal{L}(T_V, v_1)$, which can be determined in $O(\log n)$ time (e.g., by binary search) after $\mathcal{L}(T_V, v_1)$ is computed in the preprocessing.

Hence, the total time for answering the query is $O(\log h \log n)$.

In the following, by making use of the above result, we consider the most general case where t can be any point in \mathcal{P} . We first present the result for the simple polygon case.

Lemma 22. *For any simple polygon P of m vertices and a source point s in P , after $O(m)$ time preprocessing, we can answer each SP-segment-intersection query in $O(\log m)$ time.*

Proof. Given any query segment τ and a point t in P , the query asks for the intersection between τ and the shortest path $\pi(s, t)$ from s to t in P (or report none if there is no intersection).

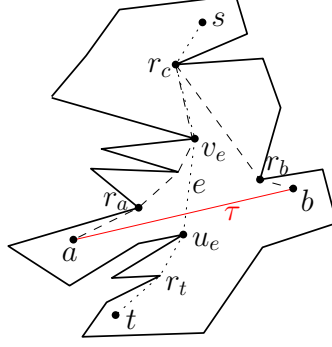


Fig. 18. Illustrating an example where $\pi(s, r_t)$ intersects the interior of τ .

In the preprocessing, we compute the shortest path tree $SPT(s)$ and shortest path map $SPM(s)$ from s in P , which can be done in $O(m)$ time [17]. We then build a point location data structure on $SPM(s)$ in $O(n)$ time [14,25]. Further, we compute the canonical cycle $\mathcal{C}(SPT(s))$ in $O(m)$ time.

Let r_t be the root of the cell of $SPM(s)$ containing t such that $\pi(s, t)$ contains $\overline{r_t t}$. We first check whether $\overline{r_t t}$ intersects τ . If yes, we return the intersection. Otherwise, we proceed to compute the intersection between τ and the shortest path $\pi(s, r_t)$ from s to r_t .

Let a and b be the two endpoints of τ , respectively. We first check whether a is on $\pi(s, r_t)$, as follows. If $a \in \pi(s, r_t)$, then a must be on an edge e of $\pi(s, r_t) \subseteq SPT(s)$, and further, r_t must be a descendent of v_e , where v_e is the endpoint of e farther to s in $\pi(s, r_t)$. Therefore, to check whether a is on $\pi(s, r_t)$, we can use the following approach. First, we determine whether a is on an edge of $SPT(s)$, which can be done in $O(\log m)$ time by a point location query on the decomposition of $SPM(s)$ by the edges of $SPT(s)$. If a is not on an edge of $SPT(s)$, then we know that a cannot be in $\pi(s, r_t)$. Otherwise, we proceed on determining whether r_t is a descendent of v_e . To this end, observe that r_t is a descendent of v_e if and only if the lowest common ancestor of v_e and r_t in $SPT(s)$ is v_e , which can be computed in $O(1)$ time after $O(m)$ time preprocessing on $SPT(s)$ [3,18].

Hence, we can check whether a is in $\pi(s, r_t)$ in $O(\log m)$ time. Similarly we can check whether b is in $\pi(s, r_t)$ in $O(\log m)$ time. If either a or b is on $\pi(s, r_t)$, then we stop the algorithm and return it as an intersection of τ and $\pi(s, t)$. Below, we assume neither a nor b is in $\pi(s, r_t)$. Thus, our goal is to compute the intersection between $\pi(s, r_t)$ and the interior of τ .

Let r_a be the root of the cell of $SPM(s)$ containing a . Define r_b similarly. Let r_c be the lowest common ancestor of r_a and r_b in $SPM(s)$ (e.g., see Fig. 18), which can be found in constant time by a lowest common ancestor query. Let F denote the funnel that is the region of P bounded by $\pi(r_c, a)$, $\pi(r_c, b)$, and \overline{ab} . Note that both $\pi(r_c, a)$ and $\pi(r_c, b)$ are convex with the convexity towards the interior of F . We assume that if we traverse from r_c counterclockwise around ∂F we will be on $\pi(r_c, a)$ before arriving at τ (otherwise we exchange the notation a and b). Observe that $\pi(s, r_t)$ intersects the interior of τ if and only if there is an edge e of $\pi(s, r_t)$ such that e intersects the interior of τ and one endpoint of e is in F and the other one is outside F (e.g., see Fig. 18). Let v_e be the endpoint of e in F and u_e be the endpoint of e outside F . Observe that such an edge e exists if and only if r_t is between r_a and r_b counterclockwise in the circular list $\mathcal{C}(SPT(s))$, which can be determined in $O(\log m)$ time by binary search on the list.

Further, if such an edge $e = \overline{u_e v_e}$ exists, then we further compute the intersection $e \cap \tau$. To determine the edge e , we first find the vertex v_e as follows. We find the lowest common ancestor of r_t and r_a , denoted by v_1 . If v_1 is not r_c , then v_1 must be on $\pi(r_c, r_a)$ and v_e is v_1 . Otherwise, the

lowest common ancestor of r_t and r_b is v_e . After v_e is found, e is the first edge in the shortest path $\pi(v_e, r_t)$ from v_e to v_t , which can be found in $O(\log m)$ time using a two-point shortest path query on the vertex pair (v_e, r_t) with $O(m)$ time preprocessing [16,19]. \square

Combining all our results above, the following lemma gives our final result.

Lemma 23. *Given $SPM(s)$, we can build a data structure of $O(n \log h)$ size in $O(n \log h)$ time that can answer each SP-segment-intersection query in $O(\log h \log n)$ time.*

Proof. In the preprocessing, we build the tree T_1 , which takes $O(n \log h)$ time and space. For each cell Δ of the decomposition \mathcal{D} , since it is a simple polygon, we build the data structure in Lemma 22 with respect to each super-root of Δ ; this takes $O(n)$ time and space in total.

Given τ and t , our query algorithm works as follows. We first determine the cell Δ of \mathcal{D} that contains t . We also determine the super-root r of Δ such that $\pi(s, t) = \pi(s, r) \cup \pi(r, t)$. All this can be done in $O(\log n)$ time. Note that r is a vertex in T_V . Hence, we can compute an intersection between τ and $\pi(s, r)$ in $O(\log h \log n)$ time using the tree T_1 . If there is an intersection, we return it and stop the algorithm. Otherwise, we compute an intersection between τ and $\pi(r, t)$ in the cell Δ . To this end, we first compute the at most two sub-segments of $\tau \cap \Delta$ by using the ray-shooting queries inside and outside Δ . For this, in the preprocessing, for each cell Δ of \mathcal{D} , we compute ray-shooting data structures on both the inside and outside of Δ (e.g., by the similar techniques as in Lemma 20). Computing these ray-shooting data structure on all cells of \mathcal{D} takes $O(n)$ time. Then, for each sub-segment τ' of $\tau \cap \Delta$, we compute the intersection (if any) between τ' and $\pi(r, t)$ in $O(\log n)$ time by Lemma 22. Hence, the overall query algorithm runs in $O(\log h \log n)$ time.

The lemma thus follows. \square

4.6.3 The \mathcal{R} -Region Range Queries

In the following, we give our data structure for answering the \mathcal{R} -region queries. Specifically, given a range $[i, j]_R$ of indices of the regions of \mathcal{R} and an extended-window $\tau \in W$, the query asks for the ccw-largest index $r \in [i, j]_R$ such that τ crosses the region boundary ∂R_r (or report none if such an index does not exist). We actually consider a more general query where τ can be any segment in \mathcal{P} (not necessarily in W). Our goal is to show the following result.

Lemma 24. *Given $SPM(s)$, we can build a data structure in $O(n \log h)$ time and space such that each \mathcal{R} -region range query can be answered in $O(\log h \log n)$ time.*

Recall that for each region $R_r \in \mathcal{R}$, its boundary ∂R_r consists of three portions: $\pi(s, v_r)$, $\pi(s, v_{r+1})$, and α_r .

Recall that $\mathcal{L}_l(T_v, v_1) = \{v_1, v_2, \dots, v_{h^*}\}$. We build a complete binary search tree T_2 as follows. Like T_1 in Section 4.6.2, the leaves of T_2 from left to right correspond to v_1, v_2, \dots, v_{h^*} . For each node u of T_2 , we construct the same auxiliary data structure $P(u)$ as in T_1 . In addition, we build another auxiliary data structure $U(u)$ for each internal node u of T_2 as follows.

We use $T_2(u)$ to denote the subtree of T_2 rooted at u and use $S(u)$ to denote the set of the leaves of $T_2(u)$. As in T_1 in Section 4.6.2, each point of V corresponds to a leaf of $S(u)$ and is also a leaf of T_V . Let p_u be the point of the path $P(u)$ in T_V that is farthest from s . In the case where $P(u)$ is empty, let p_u be $p_{u'}$ for the parent u' of u if $u \neq s$ and $p_u = s$ otherwise. Note that p_u is a node of T_V . Let U be the union of the paths of T_V from p_u to all leaves of $S(u)$ in T_V ,

excluding the sub-path from s to p_u in T_V . It is not difficult to see that U is actually a subtree of T_V . Recall that the points of $S(u)$ are consecutive in the list $\mathcal{L}_l(T_v, v_1) = \{v_1, v_2, \dots, v_{h^*}\}$. Let $S(u)$ be v_a, v_{a+1}, \dots, v_b with $1 \leq a \leq b \leq h^*$. If $a < b$ (i.e., u is not a leaf), for each $c \in [a, b-1]$, recall that α_c belongs to ∂R_c and α_c is either a bisector super-curve or a chain of obstacle edges, and we add α_c to U if α_c is a bisector super-curve. The resulting U is $U(u)$. Note that $U(u)$ is connected since every point of $U(u)$ has a path on $U(u)$ connecting to the point p_u . We consider $U(u)$ as a subdivision of the plane by all edges of $U(u)$, without considering the obstacles of \mathcal{P} .

We claim that each cell (excluding the outer unbounded one) of $U(u)$ is simply connected. Indeed, if $U(u)$ does not contain any bisector super-curve α_c , then $U(u)$ is a connected subtree of T_V and thus there is only one cell, which is the outer unbounded one. If $U(u)$ contains a bisector super-curve α_c for some $c \in [a, b-1]$, then α_c along with $\pi(p_u, v_c)$ (which is also the path from p_u to v_c in T_V and is in $U(u)$) and $\pi(p_u, v_{c+1})$ forms a closed cell C of $U(u)$. Note that C is also a cell in the decomposition \mathcal{D}' . Also, for any closed cell C' of $U(u)$ (i.e., C' is not the outer unbounded one), C' must be formed by a bisector super-curve in $U(u)$ as discussed above. Therefore, each closed cell of $U(u)$ is simply connected.

For each closed cell C of $U(u)$, we build a ray-shooting data structure. Although C has a bisector super-curve, which consists of hyperbolic curves instead of line segments, Melissaratos and Souvaine [26] showed that we can still build a ray-shooting data structure for C in linear time and space such that each query can be answered in logarithmic time¹.

For the outer cell C of $U(u)$, we can use the similar approach as Lemma 20 to preprocess it in linear time such that each ray-shooting query on C can be answered in logarithmic time.

In addition, recall that α_{h^*} connects v_{h^*} and v_1 . If α_{h^*} is a bisector super-curve, then we build a ray-shooting data structure for α_{h^*} [26].

This finishes the description of our data structure T_2 .

Lemma 25. *The space of T_2 is $O(n \log h)$ and T_2 can be built in $O(n \log h)$ time.*

Proof. First of all, the auxiliary data structures $P(u)$ on all nodes u of T_2 can be built in $O(n \log h)$ time and space as in Lemma 21. In the following, we focus on the second auxiliary data structure $U(u)$. To analyze the total space, we first show that each edge e of T_V can be in $U(u)$ for at most two nodes u in each level of T_2 .

Indeed, assume to the contrary that there are three such nodes. Since the points of V whose paths from s in T_V that contain e must be consecutive in the list $\{v_1, v_2, \dots, v_{h^*}\}$ (and thus in the consecutive leaves of T_2), by the similar analysis as in Lemma 21, we can find two nodes u_1 and u_2 sharing the same parent such that e is contained in both $U(u_1)$ and $U(u_2)$. But this implies that e must be stored in $P(u)$ for a proper ancestor u of u_1 (or u_2). This further implies that e cannot be stored in either $U(u_1)$ or $U(u_2)$.

Hence, each edge e of T_V can be in $U(u)$ for at most two nodes u in the same level of T_2 . Consequently, each edge of T_V is contained in $U(u)$ for at most $O(\log h)$ nodes u of T_2 , as the height of T_2 is $O(\log h)$.

Next we show that for each bisector super-curve α_c , it is stored in $U(u)$ for at most $O(\log h)$ nodes u of T_2 . Recall that the two endpoints of α_c are two leaves v_c and v_{c+1} of T_2 . Notice that α_c is in $U(u)$ if and only if $[c, c+1] \subseteq [l_u, r_u]$, where l_u (resp., r_u) is the index of the leftmost (resp., rightmost) leaf in the subtree $T_2(u)$. Clearly, $[c, c+1] \subseteq [l_u, r_u]$ if and only if u is in the path from the root to the lowest common ancestor of v_c and v_{c+1} , and there are $O(\log h)$ such nodes u .

¹ In fact, since each bisector edge of $SPM(s)$ is a convex curve, C is naturally a splinegon [26].

Since the total size of all bisector super-curves is $O(n)$, the space of $U(u)$ in T_2 used to store the bisector super-curves is $O(n \log h)$.

Combining the above discussions, the size of T_2 is $O(n \log h)$.

For each node u of T_2 , constructing $U(u)$ can be done in linear time in the size of $U(u)$ as follows. Let v_a, v_{a+1}, \dots, v_b be the leaves in $T_2(u)$. We consider the paths from p_u to these leaves in T_V one by one in a bottom-up manner. Initially we let $U(u)$ contain the only path $\pi(p_u, v_a)$. In general, suppose $\pi(p_u, v_{c-1})$ has been considered (initially, $c-1 = a$). Then we process $\pi(p_u, v_c)$ as follows. We traverse on $\pi(p_u, v_c)$ from v_c to p_u in T_V until we meet an obstacle vertex that is on the current $U(u)$, and then add all traversed edges of $\pi(p_u, v_c)$ to $U(u)$. We continue the algorithm as above until $\pi(p_u, v_b)$ is processed. Finally, for each $c \in [a, b-1]$ (if $a < b$), if α_c is a bisector super-curve, then we add α_c to $U(u)$. The above algorithm constructs $U(u)$ in linear time.

Then, we construct the ray-shooting data structures for the cells of $U(u)$, which can also be done in linear time in the size of $U(u)$.

Since the total size of $U(u)$ of all nodes u of T_2 is $O(n \log h)$, the total time for constructing the second auxiliary data structures is $O(n \log h)$. Therefore, T_2 can be computed in $O(n \log h)$ time. \square

By using the tree T_2 , the following lemma gives our query algorithm, which proves Lemma 24.

Lemma 26. *Each \mathcal{R} -region range query can be answered in $O(\log h \log n)$ time.*

Proof. Given a range $[i, j]_R$ of indices of the regions of \mathcal{R} and a segment $\tau \in \mathcal{P}$, we want to compute the ccw-largest index $r \in [i, j]_R$ such that τ crosses the boundary ∂R_r (if no such index r exists, then we return none). Let r^* be the sought index.

Recall that both $i \leq j$ and $i > j$ are possible. We first consider the case where $i \leq j$. In this case, $[i, j]_R$ consists of $\{i, i+1, \dots, j\}$. We begin with finding the lowest common ancestor of the two leaves v_i and v_j in T_2 , denoted by w . Our algorithm consists of four procedures.

The first procedure. The first procedure considers the nodes in the path of T_2 from the root to w . For each node u in the path, we check whether τ crosses $P(u)$ by a ray-shooting query. If yes, then τ crosses the shortest path $\pi(s, v_j)$ and thus crosses ∂R_j . Hence, we can simply return $r^* = j$ and stop the algorithm. Otherwise, we proceed on the next node until w is considered.

After w is considered, if r^* is not found, then we go to the second procedure.

The second procedure. The second procedure considers the nodes in the path of T_2 from u_j up to w in a bottom-up fashion. For each node u , there are three cases.

1. If $u = w$, we stop the second procedure and go to the third procedure.
2. If $u = u_j$, then we check whether τ intersects $P(u)$ by calling a ray-shooting query. If there is an intersection, we return $r^* = j$. Otherwise, we proceed on the parent of u .
3. Suppose u is neither u_j nor w .
 If u_j is in the left sub-tree of u , then we check whether τ intersects $P(u)$ by a ray-shooting query. If there is an intersection, then we return $r^* = j$. Otherwise, we proceed on the parent of u .
 If u_j is in the right sub-tree of u , then we first check whether τ intersects $P(u)$. If yes, then we return $r^* = j$. Otherwise, let u' be the left child of u (if u does not have a left child, then we proceed on the parent of u). We proceed as follows.

We check whether τ intersects $P(u')$. If yes, we return r^* as the rightmost index of the leaves in the subtree $T_2(u')$. Otherwise, we check whether τ intersects $U(u')$ by first locating the cell C of $U(u')$ containing an endpoint of τ and then calling a ray-shooting query on C . If not, we proceed on the parent of u (not u'). Otherwise, we set $u = u'$ and go to the fourth procedure.

The third procedure. In this procedure, we consider the vertices on the path of T_2 from the left child of w down to u_i , which is symmetric to the second procedure. For each node u , there are two cases.

1. If $u \neq u_i$, we first check whether τ intersects $P(u)$ by a ray-shooting query. If yes, we return the index of the rightmost leaf of $T_2(u)$ as r^* . Otherwise, if u_i is at the right subtree of u , then we proceed on the right child of u .
If u_i is at the left subtree of u , let u' be the right child of u (if u does not have a right child, then we proceed on the left child of u). We first check whether τ intersects $P(u')$. If yes, we return the index of the rightmost leaf of $T_2(u')$ as r^* . Otherwise, we check whether τ intersects $U(u')$. If not, we proceed on the left child of u . Otherwise, we set $u = u'$ and go to the fourth procedure.
2. If $u = u_i$, then we check whether τ intersects $P(u)$. If yes, we return $r^* = i$. Otherwise, we return none, i.e., τ does not intersect ∂R_r for any $r \in [i, j]_R$.

The fourth procedure. In the fourth procedure, we have a vertex u of T_2 such that τ does not intersect $P(u)$ but intersects $U(u)$. Starting from u , the procedure works as follows. If u is a leaf, then we simply return the index of the leaf as r^* . Otherwise, let u' be the right child of u . If τ intersects $P(u')$, then we return r^* as the index of the rightmost leaf of $T_2(u')$. Otherwise, we check whether τ intersects $U(u')$. If yes, we set u to u' and proceed as above. Otherwise, we set u to the left child of u and proceed as above.

For the running time of the algorithm, observe that the algorithm only visits $O(\log h)$ vertices of T_2 and makes $O(\log h)$ ray-shooting queries as the height of T_2 is $O(\log h)$. Each ray-shooting query is either on $P(u)$ or $U(u)$ for some node u of T_2 , which runs in $O(\log n)$ time. Hence, the total time of the algorithm is $O(\log h \log n)$.

The above gives the query algorithm for the case $i \leq j$. If $i > j$, then the index range $[i, j]_R$ consists of $\{i, i+1, \dots, h^*, 1, 2, \dots, j\}$. For this case, we first apply the above query algorithm on the range $[1, j]_R$. If the query does not return none, then we return r^* as the answer to the original query on $[i, j]_R$. Otherwise, if α_{h^*} is a bisector super-curve, then we check whether τ intersects α_{h^*} by a ray-shooting query; if there is an intersection, then we return $r^* = h^*$. Otherwise, we apply the above query algorithm on the range $[i, h^*]$, and the result of the query is the answer to the original query on $[i, j]_R$. The total time of the query algorithm is still $O(\log h \log n)$.

The lemma thus follows. \square

4.7 Wrapping Things Up

We summarize our overall result in the following theorem.

Theorem 2. *Given $SPM(s)$, we can build a data structure of $O(n \log h + h^2)$ size in $O(n \log h + h^2 \log h)$ time, such that each quickest visibility query can be answered in $O((K + h) \log h \log n)$ time, where K is the size of the visibility polygon of the query point q .*

Proof. In the preprocessing, we compute the visibility polygon query data structure in [9] for computing $Vis(q)$, which is of $O(n + h^2)$ size and can be built in $O(n + h^2 \log h)$ time. The rest of the preprocessing work includes building the decomposition \mathcal{D} and the segment query data structure as in Section 3, performing the preprocessing in Lemmas 8, 10, 16, 23, and 24; these work takes $O(n \log h)$ time and space in total.

Given any query point q , we first compute $Vis(q)$ in $O(K \log n)$ time by the query algorithm in [9]. Then, we obtain the extended window set W . Let $k = |W|$, which is $O(K)$. Next, we compute a closest point q^* on a segment of W in $O(k \log h \log n)$ time. To this end, we compute a set S of $O(k)$ candidate points as follows. We first add q, q_1, \dots, q_k to S . Then, we compute the closest point q_0^* of $\overline{u_0 q_0}$ and add q_0^* to S . Next we compute the point q_l^* in $O((k + h) \log h \log n)$ time by using our pruning algorithm in Sections 4.3 and 4.5. By a symmetric algorithm, we can also compute q_r^* . We add both q_l^* and q_r^* to S . By our analysis, q^* must be one of the points of S . Since $|S| = O(k)$, we can find q^* in S in additional $O(k \log n)$ time by using the shortest path map $SPM(s)$. \square

In fact, we have the following more general result, which might have independent interest.

Corollary 1. *Given $SPM(s)$, we can build a data structure of $O(n \log h)$ size in $O(n \log h)$ time, such that given $k = O(n)$ segments in \mathcal{P} intersecting at the same point, we can compute a shortest path from s to all these segments in $O((k + h) \log h \log n)$ time.*

Proof. The preprocessing step is the same as in Theorem 2 except that the visibility polygon query data structure [9] is not necessary any more. Hence, the total preprocessing time and space is $O(n \log h)$.

Given a set S of k segments intersecting at the same point, denoted by p , we break each segment at p to obtain two segments and we still use S to denote the new set of at most $2k$ segments. Next we compute a closest point p^* on the segments of S . To do so, we can apply the same algorithm as in Theorem 2 for computing q^* on the extended-windows of W . Indeed, the only key property of the segments of W we need is that all segments of W have a common endpoint at q . Now that all segments of S have a common endpoint p , the same algorithm still works (some degenerate cases may happen, but can be handled easily). \square

5 The Quickest Visibility Queries: The Improved Result

In this section, we reduce the query time of Theorem 2 to $O(h \log h \log n)$, independent of K . The key idea is the following. First, we show that for any query point q , there exists a subset $\mathcal{S}(q)$ of $O(h)$ windows such that a closest point q^* is on a segment of $\mathcal{S}(q)$. Second, we give an algorithm that can compute $\mathcal{S}(q)$ in $O(h \log n)$ time, without computing $Vis(q)$. Our idea relies on the extended corridor structure [8,9,11] and modifying the query algorithm for computing $Vis(q)$ in [9].

Below we first review the extended corridor structure in Section 5.1. We then introduce the set $\mathcal{S}(q)$ in Section 5.2. Finally we present our algorithm for computing $\mathcal{S}(q)$ in Section 5.3.

5.1 The Extended Corridor Structure

The corridor structure has been used for solving shortest path problems, e.g., [7,23]. Later some new concepts such as “bays,” “canals,” and the “ocean” were introduced, e.g., [8,11], referred to as the “extended corridor structure”. We review it here for the completeness of this paper and also for introducing the notation that will be needed later.

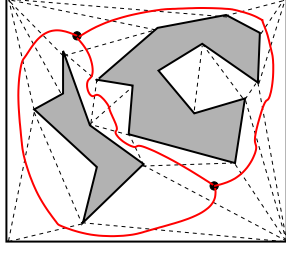


Fig. 19. Illustrating a triangulation of the free space among two obstacles and the corridors (with red solid curves). There are two junction triangles indicated by the large dots inside them, connected by three solid (red) curves. Removing the two junction triangles results in three corridors.

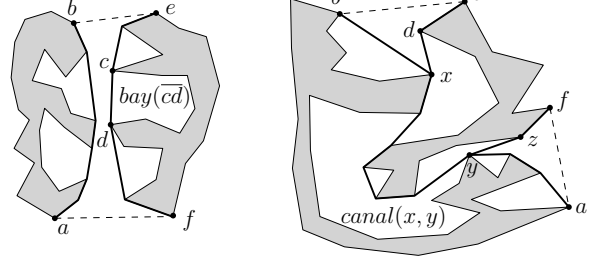


Fig. 20. Illustrating an open hourglass (left) and a closed hourglass (right) with a corridor path connecting the apices x and y of the two funnels. The dashed segments are diagonals. The paths $\pi(a, b)$ and $\pi(e, f)$ are marked by thick solid curves. A bay with gate \overline{cd} (left) and a canal with gates \overline{xd} and \overline{yz} (right) are also shown.

Let $Tri(\mathcal{P})$ denote an arbitrary triangulation of \mathcal{P} . Each edge of $Tri(\mathcal{P})$ that is not an obstacle edge of \mathcal{P} is called a (*triangulation*) *diagonal*. Let $G(\mathcal{P})$ be the (planar) dual graph of $Tri(\mathcal{P})$ (i.e., each triangle defines a node and two triangles that share a diagonal define an edge). The degree of each node in $G(\mathcal{P})$ is at most three. Using $G(\mathcal{P})$, we compute a planar 3-regular graph, denoted by G^3 (the degree of each node in G^3 is three), possibly with loops and multi-edges, as follows. First, remove every degree-one node from $G(\mathcal{P})$ together with its incident edge; repeat this process until no degree-one node remains. Second, remove every degree-two node from $G(\mathcal{P})$ and replace its two incident edges by a single edge; repeat this process until no degree-two node remains. The resulting graph is G^3 (see Fig. 19), which has $O(h)$ faces, nodes, and edges [23]. Each node of G^3 corresponds to a triangle of $Tri(\mathcal{P})$, which is called a *junction triangle*. Removing all junction triangles results in $O(h)$ *corridors* (defined below), each of which corresponds to an edge of G^3 .

The boundary of a corridor C consists of four parts (see Fig. 20): (1) A boundary portion of \mathcal{P} from a point a to a point b ; (2) a diagonal of a junction triangle from b to e ; (3) a boundary portion of \mathcal{P} from e to a point f ; (4) a diagonal of a junction triangle from f to a . The above (1) and (3) are called the two *sides* of C . The corridor C is a simple polygon.

Let $\pi(a, b)$ (resp., $\pi(e, f)$) be the shortest path from a to b (resp., e to f) in C . The region H_C bounded by $\pi(a, b)$, $\pi(e, f)$, and the two diagonals \overline{be} and \overline{fa} is called an *hourglass*, which is *open* if $\pi(a, b) \cap \pi(e, f) = \emptyset$ and *closed* otherwise (see Fig. 20). If H_C is open, then both $\pi(a, b)$ and $\pi(e, f)$ are convex chains and are called the *sides* of H_C ; otherwise, H_C consists of two “funnels” and a path $\pi_C = \pi(a, b) \cap \pi(e, f)$ joining the two apices of the two funnels, called the *corridor path* of C . Each side of every funnel is also a convex chain.

The triangulation $Tri(\mathcal{P})$ can be computed in either $O(n \log n)$ time or $O(n + h \log^{1+\epsilon} h)$ time for any constant $\epsilon > 0$ [2]. After $Tri(\mathcal{P})$ is produced, computing all corridors and hourglasses takes $O(n)$ time.

Let \mathcal{M} be the union of all $O(h)$ junction triangles, open hourglasses, and funnels. We call \mathcal{M} the *ocean*, which is a subset of \mathcal{P} . Since the sides of open hourglasses and funnels are all convex, the boundary $\partial\mathcal{M}$ of \mathcal{M} consists of $O(h)$ convex chains with a total of $O(n)$ vertices.

The space of \mathcal{P} not in \mathcal{M} , i.e., $\mathcal{P} \setminus \mathcal{M}$, consists of two types of regions: *bays* and *canals*, defined as follows. Consider the hourglass H_C of a corridor C .

We first discuss the case where H_C is open (see Fig. 20). The boundary of H_C has two sides. Let c and d be any two consecutive vertices on one side of H_C such that \overline{cd} is not an obstacle edge

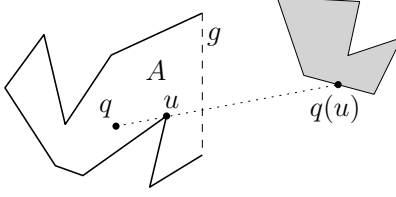


Fig. 21. Illustrating an outer-bay window $w_u = \overline{uq(u)}$, where q is in a bay A (g is the gate).

(see the left figure in Fig. 20). Both c and d must be on the same side of the corridor C . The region enclosed by \overline{cd} and the side of C between c and d is called a *bay*. We call \overline{cd} the *gate* of the bay, which is a common edge of the bay and \mathcal{M} .

If the hourglass H_C is closed, let x and y be the two apices of its two funnels. Consider two consecutive vertices c and d on a side of a funnel such that \overline{cd} is not an obstacle edge. If c and d are on the same side of the corridor C , then \overline{cd} also defines a bay. Otherwise, one of c and d must be a funnel apex, say, $c = x$, and we call \overline{xd} a *canal gate* (see Fig. 20). Similarly, there is also a canal gate at the other funnel apex y , say \overline{yz} . The region of C bounded by the two canal gates \overline{xd} and \overline{yz} that contains the corridor path is the *canal* of H_C .

Each bay or canal is a simple polygon. While the total number of all bays is $O(n)$, the total number of all canals is $O(h)$ since the number of corridors is $O(h)$. The two obstacle vertices of each bay/canal gate are called *gate vertices*.

5.2 Defining the Window Set $\mathcal{S}(q)$

We consider the source point s as an obstacle and build the extended corridor structure. This means that s is on the boundary of the ocean \mathcal{M} and thus is not in any bay or canal.

Consider any query point q . For any bay, if q is not in the bay, since the bay has only one gate, q cannot see any point outside the bay “through” its gate. Although a canal has two gates, the next lemma, proved in [11], gives an important property that if q is outside a canal, then q cannot see any point outside the canal through the canal (and its two gates).

Lemma 27. [11] (The Opaque Property) *For any canal, for any line segment \overline{pq} in \mathcal{P} (i.e., p is visible to q) such that neither p nor q is in the canal. Then \overline{pq} cannot contain any point of the canal that is not on its two gates.*

Consider any window $w_u = \overline{uq(u)}$ of q defined by u , i.e., $q(u)$ is the first point on $\partial\mathcal{P}$ hit by the ray from u along the direction from q to u . Clearly, the extended-window $\overline{qq(u)}$ is locally tangent at u , i.e., the two incident obstacle edges to u must be on the same side of the supporting line of $\overline{qq(u)}$. In the following, we partition all windows of q into different types.

Recall that $\partial\mathcal{M}$ is comprised of $O(h)$ convex chains. We call w_u an *ocean window* if u is a vertex of a convex chain of $\partial\mathcal{M}$ such that $\overline{qq(u)}$ is outer tangent to that convex chain at u . Since q has at most two extended-windows outer tangent to each convex chain, q has $O(h)$ ocean windows.

Suppose w_u is not an ocean window. If $\overline{qu} \setminus \{u\}$ does not contain any point in \mathcal{M} , then \overline{qu} is in a bay/canal A . In this case, we say w_u is an *outer-bay/outer-canal window* defined by A (we use “outer” because it is possible that $w_u = \overline{uq(u)}$ contains points outside A); e.g., see Fig. 21.

If $\overline{qu} \setminus \{u\}$ contains a point q' in \mathcal{M} , then $q' \neq u$. Depending on whether u is on $\partial\mathcal{M}$, there are two cases.

If u is on $\partial\mathcal{M}$, then $\overline{q'u}$ is in \mathcal{M} this is because $\overline{q'u}$ cannot traverse through the interior of a canal due to the opaque property of Lemma 27. If we move from q' to $q(u)$ on $\overline{qq(u)}$, since w_u is not an ocean window, after we pass u , we must move into the inside of a bay/canal A , and further, regardless of whether A is a bay or a canal, we will never get out of A due to the opaque property, which implies that $w_u = \overline{uq(u)}$ must be in A . In this case, we say that w_u is an *inner-bay/inner-canal window* defined by A (we use “inner” because w_u is in A).

If u is not on $\partial\mathcal{M}$, then u is a non-gate vertex of a bay/canal A . This implies that if we move from q' to u on $\overline{qq(u)}$, we must cross a gate of A . Again, regardless of whether A is a bay or a canal, $w_u = \overline{uq(u)}$ must be in A . In this case, we also call w_u an *inner-bay/inner-canal window* (e.g., see Fig. 22 and Fig. 23).

As a summary, a window w_u may be an ocean window, an outer-bay/canal window, or an inner-bay/canal window.

A window of q is called a *closest window* if it contains a closest point q^* of $\text{Vis}(q)$.

The set $\mathcal{S}(q)$ is defined as follows. We first add all $O(h)$ ocean windows to $\mathcal{S}(q)$. We will show several observations. First, no inner-bay window can be a closest window. Second, among all inner-canal windows defined by the same canal, there are at most two that can be closest windows and we add them to $\mathcal{S}(q)$. Since there are $O(h)$ canals, $\mathcal{S}(q)$ has $O(h)$ inner-canal windows. Third, among all outer-bay windows, there are at most two that can be closest windows; we add them to $\mathcal{S}(q)$. Fourth, among all outer-canal windows, there are at most four that can be closest windows; we add them to $\mathcal{S}(q)$. This finishes the definition of $\mathcal{S}(q)$. In summary, $\mathcal{S}(q)$ has $O(h)$ ocean windows, $O(h)$ inner-canal windows, at most two outer-bay windows, and at most four outer-canal windows. Thus, the size of $\mathcal{S}(q)$ is $O(h)$.

For a window $w_u = \overline{uq(u)}$, we assume it is directed from u to $q(u)$ and also assume $\overline{qq(u)}$ is directed from q to $q(u)$.

Observation 10 *Suppose w_u is a closest window, i.e., $q^* \in w_u$. If the two obstacle edges incident to u are on the left (resp., right) side of $\overline{qq(u)}$, then the shortest path from s to q^* must be from the left (resp., right) side of w_u .*

Proof. As discussed before, $\pi(s, q^*)$ is either from the left or from the right side of w_u . Without loss of generality, we assume that the two obstacle edges incident to u are on the left side of $\overline{qq(u)}$.

Assume to the contrary that $\pi(s, q^*)$ is from the right side of w_u . Let p be a point on $\pi(s, q^*)$ infinitely close to q^* but $p \neq q^*$. Since the two obstacle edges incident to u are on the left side of $\overline{qq(u)}$, p is visible to q , i.e., $p \in \text{Vis}(q)$. Since $d(s, p) < d(s, q^*)$, q^* cannot be a closest point of $\text{Vis}(q)$, a contradiction. \square

Lemma 28. *None of the inner-bay windows is a closest window.*

Proof. Suppose $w_u = \overline{uq(u)}$ is an inner-bay window defined by a bay A . By definition, w_u is in A . Assume to the contrary that w_u is a closest window.

Without loss of generality, assume the two obstacle edges of \mathcal{P} incident to u is on the left side of $\overline{qq(u)}$ (e.g., see Fig. 22). Since both u and $q(u)$ are on the boundary of A , w_u partitions A into two sub-polygons and one of them contains the only gate g of A . Let A' be the sub-polygon that does not contain g . Observe that A' must be locally on the left side of w_u . By Observation 10, since $q^* \in w_u$, $\pi(s, q^*)$ must be from the left side of w_u , implying that p must be in the interior of A' , where p is a point on $\pi(s, q^*)$ infinitely close to q^* . Clearly, s is not in A' . Thus, $\pi(s, p)$ must cross w_u , but this is not possible since q^* is on w_u . Thus, w_u cannot be an closest window. \square

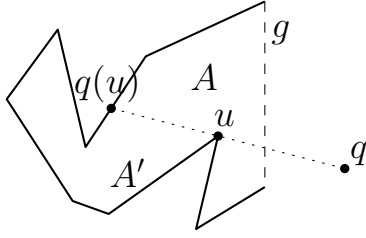


Fig. 22. Illustrating an inner-bay window $w_u = \overline{uq(u)}$ in a bay A .

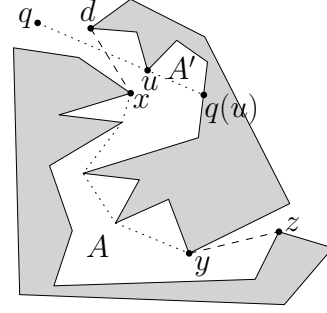


Fig. 23. Illustrating an inner-canal window $w_u = \overline{uq(u)}$ defined by a canal A with two gates \overline{xd} and \overline{yz} .

Lemma 29. *For any canal A that defines an inner-canal window w_u , if u is not an endpoint of the corridor path of A , then w_u cannot be a closest window.*

Proof. Since w_u is an inner-canal window defined by A , w_u must be in A and both u and $q(u)$ are on the boundary of A . Further, $\overline{qu(q)}$ has a point $q' \in \mathcal{M}$ and $\overline{q'u}$ crosses a gate g of A . Let $g = \overline{xd}$ such that x is the endpoint of the corridor path of A on g (e.g., see Fig. 23). Let C be the corridor that defines the canal A .

Assume without loss of generality that the two obstacle edges of \mathcal{P} incident to u are on the left side of $\overline{qu(q)}$. Since u is not x , according to the results in [11] (see the proof of Lemma 3) that u and $q(u)$ must be on the same side of C that contains d (e.g., see Fig. 23). This implies that w_u partitions A into two sub-polygons one of which contains both gates of A , and let A' be the sub-polygon that does not contain the gates. Then, as in the proof of Lemma 28, A' must be locally on the left side of w_u , and by the similar analysis we can show that w_u cannot be a closest window. \square

Since each canal has one corridor path, the preceding lemma implies that every canal can define at most two inner-canal windows that are possibly closest windows.

Consider a bay A with gate g that defines an outer-bay window w_u . By definition, \overline{qu} is in A . Let u_1 be the vertex of A such that $\overline{qu_1}$ is in the shortest path in A from q to an endpoint of g ; similarly, define u_2 with respect to the other endpoint of g .

Lemma 30. *If w_u is an outer-bay window defined by A and u is neither u_1 nor u_2 , then w_u cannot be a closest window.*

Proof. By the definitions of u_1 and u_2 , since A is a simple polygon and u is neither u_1 nor u_2 , $q(u)$ must be in $\partial A \setminus \{g\}$. Hence, the window w_u partitions A into two sub-polygons and one of them contains g . Let A' be the sub-polygon that does not contain g . Then, by using the same analysis as in Lemma 28, w_u cannot be a closest window. \square

Consider a canal A that defines an outer-canal window w_u . This case is similar to the above bay case except that we need to consider both gates of A . Again, \overline{qu} is in A . Define u_1, u_2, u_3 , and u_4 similarly as in the bay case but with respect to the four gate vertices of A , respectively.

Lemma 31. *If w_u is an outer-bay window defined by A and u is not in $\{u_1, u_2, u_3, u_4\}$, then w_u cannot be a closest window.*

Proof. By the definitions of u_i for $1 \leq i \leq 4$, since A is a simple polygon and $u \notin \{u_1, u_2, u_3, u_4\}$, $q(u)$ must be in ∂A and $q(u)$ is not on a gate of A . Further, it can be verified that the window w_u partitions A into two sub-polygons and one of them contains both gates of A . Let A' be the sub-polygon that does not contain the gates of A . Then, by using the same analysis as in Lemma 28, w_u cannot be a closest window. \square

The above discussions lead to the following lemma.

Lemma 32. *Given any query point q , there is a set $\mathcal{S}(q)$ of windows of q such that $|\mathcal{S}(q)| = O(h)$ and $\mathcal{S}(q)$ contains a closest window.*

5.3 Computing the Window Set $\mathcal{S}(q)$

In this section we present our algorithm for computing $\mathcal{S}(q)$, by modifying the query algorithm in [9] for computing $\text{Vis}(q)$. Our result is summarized in the following lemma.

Lemma 33. *With $O(n + h^2 \log h)$ time and $O(n + h^2)$ space preprocessing, given any query point q in \mathcal{P} , we can compute the set $\mathcal{S}(q)$ in $O(h \log n)$ time.*

We first do the same preprocessing as in [9], which takes $O(n + h^2 \log h)$ time and $O(n + h^2)$ space. In the following, we give our query algorithm for computing $\mathcal{S}(q)$. Depending on whether q is in the ocean \mathcal{M} , a bay, or a canal, there are three cases. In each case, we will first briefly review the algorithm in [9] for computing $\text{Vis}(q)$ and then modify it to compute $\mathcal{S}(q)$.

5.3.1 The Ocean Case

Suppose q is in \mathcal{M} . The algorithm in [9] first computes the region of \mathcal{M} that is visible to q , denoted by $\text{Vis}(q, \mathcal{M})$, which is also the visibility polygon of q in \mathcal{M} due to the opaque property of canals. Then, the algorithm computes the region in all bays and canals visible to q . To this end, it traverses on the boundary of $\text{Vis}(q, \mathcal{M})$. If a gate g of a bay/canal A is encountered, then the region of A visible to q through e is computed, where e is a maximal portion of g on the boundary of $\text{Vis}(q, \mathcal{M})$. The visible regions computed above for all such e 's are pairwise disjoint. Hence, $\text{Vis}(q)$ is a trivial union of $\text{Vis}(q, \mathcal{M})$ and the visible regions in all bays and canals.

We modify the above algorithm to compute $\mathcal{S}(q)$, as follows.

The algorithm in [9] computes $\text{Vis}(q, \mathcal{M})$ by using the visibility complex [29,30]. More specifically, it uses the approach of crossing faces [30] such that all rays originating from q in the plane define a curve γ in the visibility complex and each intersection of γ and the boundary of a cell of the visibility complex corresponds to an outer tangent in \mathcal{M} from q to a convex chain of $\partial \mathcal{M}$. Note that such tangents correspond exactly to our ocean windows. If we traverse the curve γ in the visibility complex, each such intersection can be computed in $O(\log n)$ time. Hence, if there are h' convex chains of $\partial \mathcal{M}$ that are visible to q , then the endpoints of the maximal sub-chains ξ of these convex chains that are visible to q can be computed in $O(h' \log n)$ time by using the approach of crossing faces. Note that $h' = O(h)$ [9]. After this, all ocean windows are computed.

Remark. Traversing each such sub-chain ξ can explicitly construct $\text{Vis}(q, \mathcal{M})$. But for our problem of computing $\mathcal{S}(q)$, we can avoid this step; indeed, this is part of the reason our algorithm avoids the $\Omega(K)$ time.

Next, we compute other windows of $\mathcal{S}(q)$. Since q is in \mathcal{M} , $\mathcal{S}(q)$ does not have outer-bay/outer-canal windows, and we only need to compute the inner-canal windows, as follows.

The above has computed the endpoints of each such sub-chain ξ that is visible to q . If ξ does not contain any portion of any canal gate, then we simply ignore ξ . Otherwise, we need to compute the inner-canal windows through g for each canal gate g that has a portion in ξ . To this end, we need to first find these canal gates. For this, in the preprocessing step, for each convex chain C of \mathcal{M} , we maintain a list of canal gates on C by a balanced binary search tree such that given the two endpoints a and b of ξ , we can determine whether ξ contains any portion of any canal gate in $O(\log n)$ time, and if yes, report all these portions in $O(k + \log n)$ time, where k is the number of these portions. The number of such k in the entire algorithm is $O(h)$ since the total number of canal gates is $O(h)$. For each such canal gate portion e , we compute the corresponding inner-canal window (if any) as follows.

Let g be the canal gate containing e and let A be the canal. Let x be the endpoint of the corridor path of A at g . If x is not on e , then we ignore e . Otherwise, x is visible to q and x defines an inner-canal window w_u with $u = x$. Our goal is to compute $q(u)$. This can be easily done by using a ray-shooting query in A as follows. Consider the ray originating from x with direction from q to x . Using a ray-shooting query on A , we find the first point p on the boundary of A that is hit by the ray. Again, due to the opaque property of canals, p must be on an obstacle edge of \mathcal{P} , and thus $q(u) = p$. For answering each ray-shooting query in A in $O(\log n)$ time, we need to preprocess each canal for ray-shooting queries in linear time since a canal is a simple polygon, and this requires $O(n)$ time in total for all canals.

Since the number of all visible sub-chains is $O(h)$, we can compute all inner-canal windows in $O(h \log n)$ time.

In summary, we can compute the set $\mathcal{S}(q)$ in $O(h \log n)$ time for the ocean case.

5.3.2 The Bay Case

If q is in a bay A , then the algorithm in [9] for computing $\text{Vis}(q)$ first computes the region of A that is visible to q , denoted by $\text{Vis}(s, A)$. If the gate g of A does not have any point on the boundary of $\text{Vis}(s, A)$, then g is not visible to q , which further implies that no point outside the bay is visible to q and thus $\text{Vis}(s) = \text{Vis}(s, A)$. If g has a sub-segment g' on the boundary of $\text{Vis}(s, A)$, then the points of $\mathcal{P} \setminus A$ visible to q are all visible to q through g' . Next, the region $\text{Vis}(q, \mathcal{M})$ of \mathcal{M} that are visible to q through g' is computed. After $\text{Vis}(q, \mathcal{M})$ is computed, the rest of the algorithm is the same as the ocean case. Namely, by traversing the boundary of $\text{Vis}(q, \mathcal{M})$, other regions of \mathcal{P} in bays and canals visible to q can be computed.

Next we modify the above algorithm to compute $\mathcal{S}(q)$.

Since q is in A , we first compute the (at most two) outer-bay windows. Let a and b be the two endpoints of g , respectively. In the preprocessing, we compute the shortest path maps of a and b in A , respectively. We also compute a ray-shooting data structure in A . The total such preprocessing takes $O(n)$ time for all bays. Then, using the shortest path maps of a and b , the two vertices u_1 and u_2 as defined before can be computed in $O(\log n)$ time.

If $u_1 = u_2$, then consider the ray ρ originating from u_1 along the direction from q to u_1 . Let p be the first point on the boundary of A hit by ρ . Since $u_1 = u_2$, p must be on an obstacle edge of A (i.e., p is not on the gate g of A), and thus $\overline{u_1 p}$ is an outer-bay window. In fact, in this case $\overline{u_1 p}$ is the only window in $\mathcal{S}(q)$, and thus we can stop our algorithm.

If $u_1 \neq u_2$, then for each u_i with $i = 1, 2$, the intersection of g with the supporting line of $\overline{qu_i}$ is an endpoint of g' [17]. Hence, g' can be determined immediately once u_1 and u_2 are available. Similarly as in the above ocean case, the algorithm in [9] uses the approach of crossing faces to compute $\text{Vis}(q, \mathcal{M})$ through g' , which is actually a “cone” visibility query since the visibility of q in \mathcal{M} is delimited by the cone bounded by the ray from q to u_1 and the ray from q to u_2 . All rays from q in the cone define a segment γ' of the curve γ (discussed in the ocean case) in the visibility complex. To use the approach of crossing faces, the algorithm in [9] first finds the cell σ of the visibility complex that contains an endpoint of γ' , which is done in $O(\log n)$ time by a point location data structure on the visibility complex. After this, the rest of the algorithm is the same as the ocean bases. This is also the case for our problem for computing $\mathcal{S}(q)$. After locating the cell σ , we can use the crossing face approach to compute the $O(h)$ maximal sub-chains ξ of the convex chains of $\partial\mathcal{M}$ that are visible to q through g' . As in the ocean case, this will also compute all ocean windows of $\mathcal{S}(q)$. After that, we use the same approach as in the ocean case to compute all inner-canal windows. The total time is $O(h \log n)$.

Finally, we compute the two outer-bay windows defined by u_1 and u_2 . Namely, we need to compute $q(u_1)$ and $q(u_2)$. For each $i = 1, 2$, let ρ_i be the ray originating from q and along the direction from q to u_i . The above algorithm for computing the sub-chains will also determine the point p_i on $\partial\mathcal{M}$ first hit by ρ_i . If p_i is on an obstacle edge of \mathcal{P} , then p_i is $q(u_i)$. Otherwise, p_i is on a bay/canal gate g_i of a bay/canal A . Then, we use a ray-shooting query on A to find the first point p'_i on the boundary of A hit by ρ_i . Regardless of whether A is a bay or a canal, p'_i is always on an obstacle edge, and thus p'_i is $q(u_i)$. Since the ray-shooting query on A takes $O(\log n)$ time, the two outer-bay windows can be computed in $O(\log n)$ time.

In summary, the window set $\mathcal{S}(q)$ can be computed in $O(h \log n)$ time for the bay case.

5.3.3 The Canal Case

If q is in a canal A , then the algorithm is similar to the bay case with the difference that we apply the same algorithm on the two gates of the canal separately. Specifically, let $g = \overline{ab}$ be a gate of A . We first compute the vertices u_1 and u_2 with respect to a and b , respectively. Then, we apply exactly the same algorithm as in the bay case. After that, we consider the other gate of A and apply the same algorithm. Then $\mathcal{S}(q)$ is computed and the total time is $O(h \log n)$ time.

This proves Lemma 33. After $\mathcal{S}(q)$ is computed, we can apply the query algorithm of Theorem 2 (or Corollary 1) on the windows of $\mathcal{S}(q)$ to compute q^* . Thus we can obtain the following result.

Theorem 3. *Given $\text{SPM}(s)$, we can build a data structure of $O(n \log h + h^2)$ size in $O(n \log h + h^2 \log h)$ time, such that each quickest visibility query can be answered in $O(h \log h \log n)$ time.*

6 Conclusions

In this paper, we present a new data structure for answering quickest visibility queries. Our result is particularly interesting when h , the number of holes of \mathcal{P} , is relatively small. For example, when $h = O(1)$, our result matches the best result for the simple polygon case (i.e., $h = 1$) and is optimal. To achieve the result, we also solve many other problems that may be interesting in their own right. We highlight some of them below. We assume that the shortest path map $\text{SPM}(s)$ of the source point s has been given.

1. We present an algorithm that can compute a shortest path from s to τ in $O(h \log \frac{n}{h})$ time for any query segment $\tau \in \mathcal{P}$, after $O(n)$ time and space preprocessing.
2. We present an algorithm that can compute in $O(\log h \log n)$ time an intersection between τ and the shortest path $\pi(s, t)$ for any segment τ and any point t in \mathcal{P} , after $O(n \log h)$ time and space preprocessing.
3. We present an algorithm that can answer each \mathcal{R} -region range query in $O(\log h \log n)$ time, after $O(n \log h)$ time and space preprocessing.
4. We present an algorithm that can compute in $O((k + h) \log h \log n)$ time a shortest path from s to any set of $k = O(n)$ segments in \mathcal{P} that intersect at a same point, after $O(n \log h)$ time and space preprocessing.

These results are particularly interesting when h is relatively small, and at least the first three results are optimal when $h = O(1)$.

In addition, the decomposition \mathcal{D} of \mathcal{P} , the regions of \mathcal{R} , and some other techniques proposed in the paper (e.g., bundles) may find other applications as well.

References

1. E.M. Arkin, A. Efrat, C. Knauer, J.S.B. Mitchell, V. Polishchuk, G. Rote, L. Schlipf, and T. Talvitie. Shortest path to a segment and quickest visibility queries. *Journal of Computational Geometry*, 7:77–100, 2016.
2. R. Bar-Yehuda and B. Chazelle. Triangulating disjoint Jordan chains. *International Journal of Computational Geometry and Applications*, 4(4):475–481, 1994.
3. M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.
4. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry — Algorithms and Applications*. Springer-Verlag, Berlin, 3rd edition, 2008.
5. P. Bose, A. Lubiw, and J.I. Munro. Efficient visibility queries in simple polygons. *Computational Geometry: Theory and Applications*, 23(3):313–335, 2002.
6. B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.
7. D.Z. Chen and H. Wang. A nearly optimal algorithm for finding L_1 shortest paths among polygonal obstacles in the plane. In *Proc. of the 19th European Symposium on Algorithms (ESA)*, pages 481–492, 2011.
8. D.Z. Chen and H. Wang. L_1 shortest path queries among polygonal obstacles in the plane. In *Proc. of 30th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 293–304, 2013.
9. D.Z. Chen and H. Wang. Visibility and ray shooting queries in polygonal domains. *Computational Geometry: Theory and Applications*, 48:31–41, 2015.
10. D.Z. Chen and H. Wang. Weak visibility queries of line segments in simple polygons. *Computational Geometry: Theory and Applications*, 48:443–452, 2015.
11. D.Z. Chen and H. Wang. Computing the visibility polygon of an island in a polygonal domain. *Algorithmica*, 77:40–64, 2017.
12. Y.K. Cheung and O. Daescu. Approximate point-to-face shortest paths in \mathcal{R}^3 . arXiv:1004.1588, 2010.
13. Y.-J. Chiang and R. Tamassia. Optimal shortest path and minimum-link path queries between two convex polygons in the presence of obstacles. *International Journal of Computational Geometry and Applications*, 7:85–121, 1997.
14. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
15. S.D. Eriksson-Bique, J. Hershberger, V. Polishchuk, B. Speckmann, S. Suri, T. Talvitie, K. Verbeek, and H. Yıldız. Geometric k shortest paths. In *Proc. of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1625, 2015.
16. L.J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.
17. L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.

18. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.
19. J. Hershberger. A new data structure for shortest path queries in a simple polygon. *Information Processing Letters*, 38(5):231–235, 1991.
20. J. Hershberger, V. Polishchuk, B. Speckmann, and T. Talvitie. Geometric k th shortest paths: the applet. In *Video/multimedia of the 30th Annual Symposium on Computational Geometry*, 2014. <http://www.computational-geometry.org/SoCG-videos/socg14video/ksp/index.html>.
21. J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.
22. J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.
23. S. Kapoor, S.N. Maheshwari, and J.S.B. Mitchell. An efficient algorithm for Euclidean shortest paths among polygonal obstacles in the plane. *Discrete and Computational Geometry*, 18(4):377–383, 1997.
24. R. Khosravi and M. Ghodsi. The fastest way to view a query point in simple polygons. In *Proc. of the 24th European Workshop on Computational Geometry*, pages 187–190, 2005.
25. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.
26. E. Melissaratos and D. Souvaine. Shortest paths help solve geometric optimization problems in planar regions. *SIAM Journal on Computing*, 21(4):601–638, 1992.
27. J.S.B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991.
28. J.S.B. Mitchell. Shortest paths among obstacles in the plane. *International Journal of Computational Geometry and Applications*, 6(3):309–332, 1996.
29. M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudotriangulations. *Discrete and Computational Geometry*, 16(4):419–453, 1996.
30. M. Pocchiola and G. Vegter. The visibility complex. *International Journal of Computational Geometry and Applications*, 6(3):279–308, 1996.